Biblioteczka
Komputer Świat



DVD I KŚ+
- PLIKI SZKOLENIOWE
- NARZĘDZIA DLA PROGRAMISTÓW

ISO płyty i skrypty do pobrania z ksplus.pl

POZNAJ OD PODSTAW

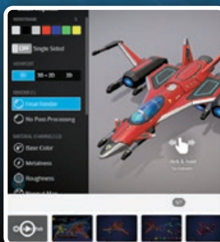
TWORZENIE GIER

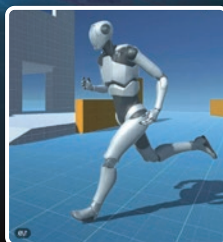
obsługa silnika **UNITY** i programowanie w **C#**

KROK PO KROKU ZAPROGRAMUJESZ:


Tworzenie scen i interakcji w grach


Generowanie i modelowanie map


Używanie gotowych obiektów


Zliczanie czasu i punktów



Z TĄ KSIĄŻKĄ E-WYDANIE GRATIS

Poniżej znajduje się płyta z kodem bonusowym dającym dostęp do e-wydania tej książki w serwisie KŚ+ (ksplus.pl) oraz pliku ISO z cyfrową wersją płyty do pobrania.

NA PŁYCIE DVD

Płyta dołączona do tej książki zawiera zestaw najlepszych darmowych narzędzi do tworzenia gier. Na DVD znajdują się też zintegrowane środowiska programistyczne, edytory kodu źródłowego oraz pliki szkoleniowe do zadań opisanych w książce.

Jeżeli brakuje płyty, poinformuj sprzedawcę lub redakcję: pomoc@komputerswiat.pl

Fot. Pikisuperstar/Freepik.com

Komputer Świat

NAJLEPSZE PROGRAMY DO TWORZENIA GIER

- silniki do tworzenia gier
- narzędzia programistyczne
- pliki szkoleniowe

PŁYTA JEST DODATKIEM DO KSIĄŻKI
KOMPUTER ŚWIAT BIBLIOTECZKA
6 | 2021

NA PŁYCIE ZNAJDZIESZ KOD DO E-WYDANIA I CYFROWEJ WERSJI PŁYTY W KSPLUS.PL

Kod bonusowy należy zarejestrować w KŚ+ (ksplus.pl)

Biblioteczka Komputer Świat

KONRAD JAGACIAK

POZNAJ OD PODSTAW

TWORZENIE GIER

obsługa silnika **UNITY** i programowanie w **C#**

ringier axel springer



AUTOR: Konrad Jagaciak

REDAKTORZY PROWADZĄCY: Rafał Kamiński, Agnieszka Al-Jawahiri

PRZYGOTOWANIE PŁYTY: Mariusz Michalski

PROJEKT OKŁADKI: Robert Dobrzyński

SKŁAD I ŁAMANIE: Mariusz Rybak

KOREKTA: Jolanta Rososińska

WYDAWCA:

RINGIER AXEL SPRINGER POLSKA Sp. z o.o.
02-672 Warszawa, ul. Domaniewska 49
tel. 12 2600200 (BOK)
www.ringieraxelspringer.pl

ISBN 978-83-8250-101-8

Warszawa 2021

BUSINESS PROJECT MANAGER: Paweł Bulwan

DRUK I OPRAWA:
Drukarnia im. Adama Półtawskiego, Kielce

EGZEMPLARZE ARCHIWALNE:
literia.pl, prenumerata.axel@qg.com

E-WYDANIA, E-PRENUMERATA:
ksplus.pl

KONTAKT:
redakcja@komputerswiat.pl

INTERNET:
komputerswiat.pl, ksplus.pl

Płyta DVD jest dodatkiem do książki

ringier
axel springer

1 Silniki gier



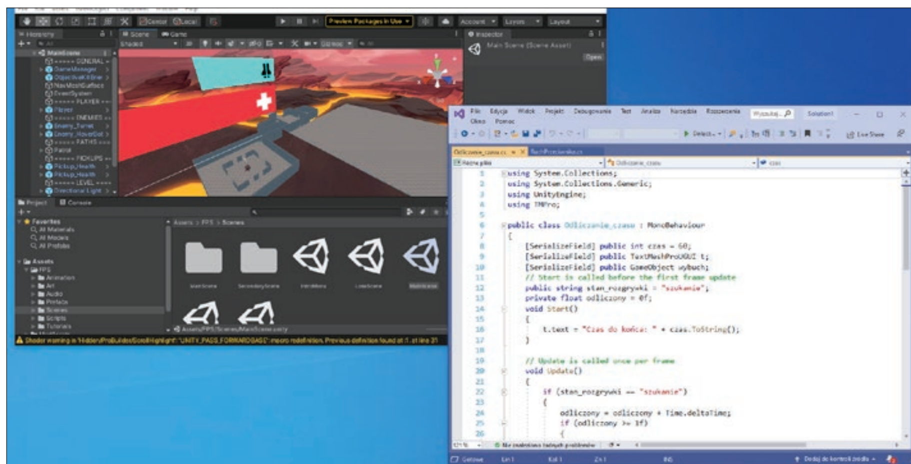
Na DVD znajdziemy silniki gier przedstawione w tym rozdziale

Główny bohater tej książki to silnik gier Unity. A co to jest, czym się charakteryzuje i do czego właściwie służy silnik gier? Tego dowiemy się z tego rozdziału

Unity (DVD-KOD: 010) – nazwa ta pojawia się w wielu miejscach. Począwszy od ogłoszeń o pracę dla programistów, a skończywszy na opisach gier, w których wymienione są technologie wykorzystane do stworzenia gry. Wiele osób jest przekonanych, że Unity to nazwa języka programowania – tak jednak nie jest. Czym zatem jest Unity?
To silnik gier, który został stworzony w języku C++ i pozwala na pisanie skryptów w języku C#. To właśnie znajomość tego języka będzie dodatkowym atutem podczas pracy z Unity. Z tej książki dowiemy się nie tylko, jak obsłużyć Unity, ale i jak korzystać z C#.

SILNIK GRY A SILNIK GRAFICZNY

Wiele osób myli silnik gry z silnikiem graficznym. Ten pierwszy to główna część kodu gry, zajmuje się interakcją między jej elementami. Ten drugi to część kodu odpowiedzialna za tworzenie grafiki 2D lub 3D i zajmuje się renderowaniem obrazu w czasie rzeczywistym, by był widoczny na wyświetlaczach; wykonuje złożone obliczenia matematyczne i przekształcenia grafiki.



Praca z Unity to wykorzystanie edytora Unity i narzędzia do edycji kodów źródłowych

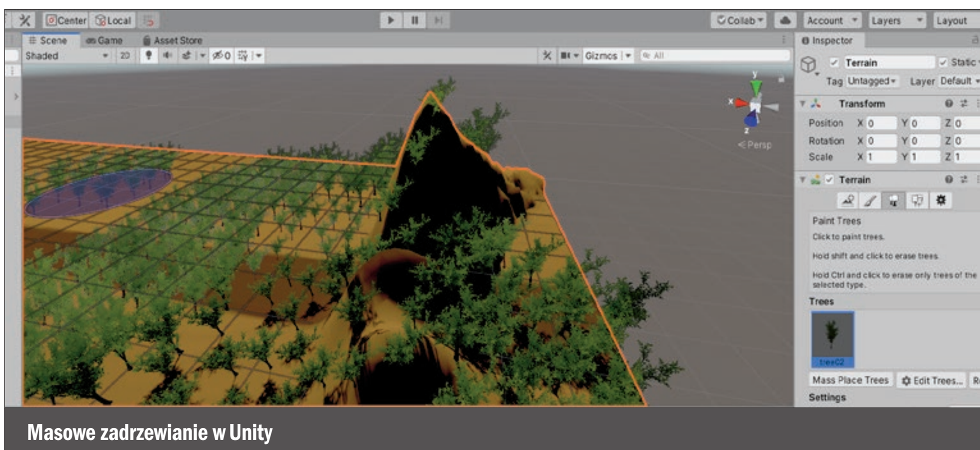
Czym są silniki gier

Silniki gier to pakiety ułatwiające tworzenie gier komputerowych. W skład takich pakietów wchodzą gotowe skrypty stanowiące główną część kodu źródłowego gry, a także narzędzia programistyczne, jak w przypadku Unity – edytor pozwalający między innymi na modelowanie świata gry poprzez ręczne ustawianie obiektów na mapie.

Silniki gier dostarczają zatem twórcom gotowe mechanizmy, które w wielu grach są powtarzalne, pozwalając na ich przebudowę i rozbudowę – tak by każda produkcja była unikalna.

Za przykład ułatwień w tworzeniu gier może posłużyć choćby edytor terenu, jaki znajdziemy w Unity. Jest to narzędzie pozwalające na zbudowanie mapy gry, oferujące między innymi funkcję masowego zadrzewiania. Dzięki temu narzędziu, tworząc grę, możemy na budowanej przez nas mapie umieszczać drzewa o podobnych rozmiarach, w losowych miejscach, z określonym zagęszczeniem. Nie musimy zastanawiać się, w jaki sposób dodać drzewo w grze – korzystamy z gotowego mechanizmu, w którym dodatkowo można ustawiać różne parametry; możemy też wybierać różne rodzaje drzew. To wszystko sprawia, że efekt użycia tej funkcji może być w każdym naszym projekcie inny.

Jak, krok po kroku, korzystać z funkcji masowego zadrzewiania, przeczytamy w dalszej części książki, tworząc własną mapę gry.



Masowe zadrzewianie w Unity

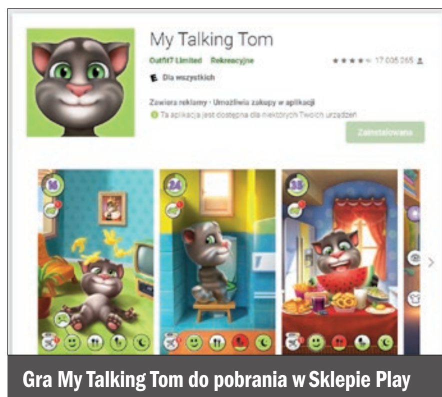
Najpopularniejsze silniki

Unity jest jednym z wielu silników gier. Na rynku dostępnych jest więcej tego typu rozwiązań. Równie popularne jest narzędzie **Unreal Engine** (DVD-KOD: 011), czyli silnik przygotowany przez firmę Epic Games. Do popularnych silników gier należy też **CryEngine** (DVD-KOD: 001), stanowiący podstawę popularnej strzelanki pierwszoosobowej – Far Cry. Unreal Engine jest darmowy dla twórców, ale od produkcji komercyjnych, które zarobi-

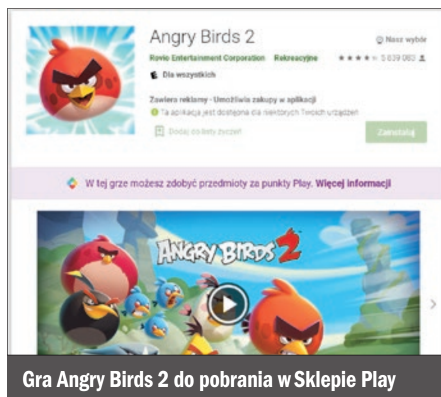
ły ponad trzy tysiące dolarów, trzeba płacić tantiemy. Starsze wersje Far Cry były całkiem darmowe, ale potem wprowadzono system płatności, w którym kupujący płaci dowolnie wybraną kwotę (Pay What You Want).
Wielu dużych producentów gier tworzy własne autorskie silniki stanowiące ich wewnątrzfirmowe narzędzia.
Jest też silnik w pełni otwarty i całkowicie darmowy – **Godot** (DVD-KOD: 003/004 32-/64-BIT), który jest udostępniany na licencji MIT, a programiści mogą nie tylko z niego korzystać, ale też włączać się w jego rozwój.
Unity (DVD-KOD: 010) początkowo było oprogramowaniem płatnym, oferowanym przez Unity Technologies twórcom gier; jego darmowa wersja była wersją demonstracyjną o okrojonych możliwościach. Ten stan rzeczy trwał przez mniej więcej 10 lat, od pojawienia się Unity w 2005 roku, aż do pojawienia się Unity w wersji 5 w 2015 roku. Po tym czasie prawie wszystkie funkcje silnika udostępniono w wersji darmowej dla twórców, których dochody nie przekraczają 100 tysięcy dolarów rocznie. To wpłynęło pozytywnie na popularność tego narzędzia, szczególnie wśród małych producentów gier. Do dziś Unity jest świetnym wyborem zarówno jako docelowe narzędzie pracy, jak i narzędzie, które pomaga uczyć się programowania i szlifować swój warsztat programistyczny.

Produkcje stworzone w Unity

W wiele produkcji stworzonych z wykorzystaniem silnika gier Unity można grać nie tylko na komputerze, ale też na innych urządzeniach, na przykład na telefonach.
Jedną z popularniejszych takich produkcji jest gra mobilna **My Talking Tom** – nazwa tej gry zapewne jest bardziej znana młodszym graczom i ich rodzicom. Głównym bohaterem gry jest kot, którym opiekujemy się jak popularnym niegdyś Tamagotchi.
Również inne hity wśród gier mobilnych zostały stworzone za pomocą Unity – na przykład gra **Angry Birds 2** (chociaż, co warte podkreślenia, nie wszystkie gry z tej serii stworzono, korzystając z Unity).
Wielbiciele nieco innych klimatów mogą kojarzyć silnik Unity z takimi tytułami, jak **Wasteland 2** czy **Wasteland 3**. Również rodzima produkcja studia CD Projekt Red – **Gwint: Wiedźmińska gra karciana** – jest oparta na silniku gier Unity. Jeśli mowa o stu-

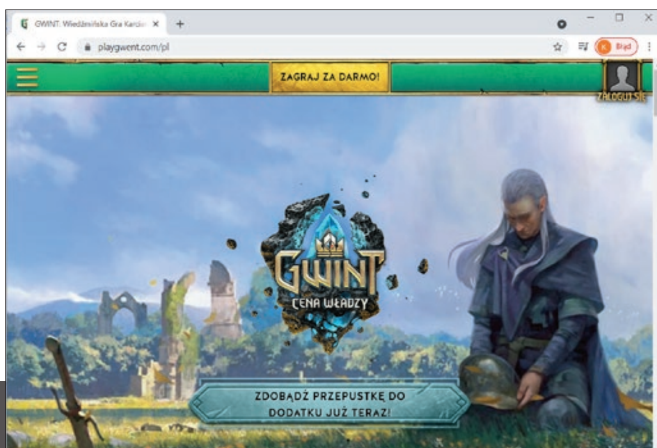


Gra My Talking Tom do pobrania w Sklepie Play



Gra Angry Birds 2 do pobrania w Sklepie Play

diu CD Projekt Red, warto wspomnieć o jego największych produkcjach, czyli serii Wiedźmin i grze Cyberpunk 2077. Na potrzeby tych produkcji studio opracowało własny silnik gier **REDengine**, stanowiący zamknięte oprogramowanie.

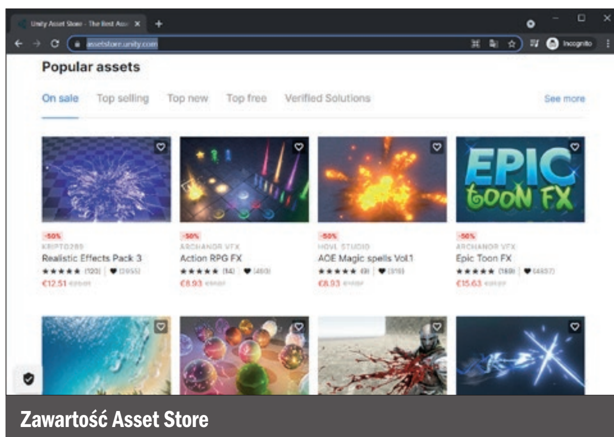


Strona internetowa gry karcianej Gwint

Asset Store

Mówiąc o ułatwieniach, jakie w procesie tworzenia gry daje nam silnik, warto też wspomnieć nie tylko o mechanizmach, ale też o konkretnych zasobach projektu, do których zaliczamy między innymi obiekty graficzne. Kiedy podczas tworzenia gry chcemy skupiać się na jej działaniu, a tworzenie grafik 3D nie jest naszą mocną stroną, możemy korzystać z udostępnianych w ramach **Asset Store** zasobów.

Asset Store to sklep z zasobami do wykorzystania w projektach. Dostęp do niego uzyskamy z poziomu strony internetowej o adresie **assetstore.unity.com** Znajdziemy tam nie tylko trójwymiarowe modele, ale też narzędzia (Tools) rozbudowujące edytor Unity o dodatkowe możliwości. W Asset Store znajdziemy nawet gotowe projekty pokazujące, jak samemu tworzyć podobne gry. Co ciekawe i warte uwagi – część zasobów w Asset Store pochodzi od twórców Unity, czyli firmy Unity Technologies, ale nie jest to jedyne źródło zasobów. Sklep pozwala twórcom na dodawanie własnych zasobów, co sprawia, że oferta Asset Store jest urozmaicona. O tym, jak korzystać z Asset Store i pobierać zasoby do późniejszego ich wykorzystania w ramach tworzonych projektów, również dowiemy się z dalszej części książki – z kolejnych jej rozdziałów.



Zawartość Asset Store



Na DVD znajdziemy Unity Hub przedstawiony w tym rozdziale

2 Zarządzanie projektami Unity

Z tego rozdziału dowiemy się, jak rozpocząć pracę z Unity – jak tworzyć nowe projekty i zarządzać dostępnymi wersjami oprogramowania

Unity – historia i najważniejsze informacje

Zanim dowiemy się, jak stworzyć pierwszą grę w Unity, i rozpoczniemy pracę nad własnymi projektami, warto poznać trochę bliżej to narzędzie.

Trochę historii

Unity Technologies, czyli firma, która stworzyła silnik Unity, powstała w Danii w 2004 roku pod nazwą Over The Edge Entertainment i zadebiutowała grą GooBall, która została opublikowana w roku 2005 na urządzenia z systemem Mac OS X.
Gra nie odniosła komercyjnego sukcesu, ale trzej założyciele firmy (David Helgason, Nicholas Francis i Joachim Ante) dostrzegli wartość w czymś innym, co powstało przy okazji. Mowa tu o narzędziach do tworzenia gier, które opracowali, aby uprościć sobie pracę. Przenieśli więc uwagę firmy na udoskonalenie silnika gier, który mogliby udostępnić innym programistom. I tak właśnie rozpoczęła się historia Unity.
Z czasem firma rozrosła się, przyjęła obecną nazwę (w 2007 roku), a jej flagowy produkt pozwalał na tworzenie gier na coraz to więcej rodzajów platform, także mobilnych, stając się jednym z najważniejszych narzędzi twórców gier.
Co ciekawe, Unity jest też wykorzystywane w branży filmowej i motoryzacyjnej dzięki temu, że umożliwia pracę nad projektami trójwymiarowymi.

GOOBALL

To wydana przez Ambrosia Software pierwsza gra stworzona przez firmę Over The Edge Entertainment, która na jej potrzeby opracowała silnik gier Unity. Gracz wciela się w niej w kosmitę, który utknął na Ziemi w urządzeniu do podtrzymywania życia wykonanym z protoplazmy.
Rozgrywka przypomina serię Super Monkey Ball, w której gracz przechyla otoczenie, co powoduje, że kula toczy się w nim, zbierając klejnoty i kierując się do mety.

Trochę problemów – i rozwiązanie

Silnik gier Unity ewoluował, a na rynku pojawiały się jego kolejne wersje. Doprowadziło to jednak do tego, że projekty, które opracowano w starszej wersji silnika, nie były zgodne z nowszymi wersjami.

By rozwijać swoje gry, twórcy zmuszeni byli często do tego, by mieć kilka wersji narzędzia. Żeby ułatwić im pracę nad projektami i zarządzanie posiadanymi wersjami silnika, stworzono nowe narzędzie – **Unity Hub** (DVD-KOD: 010).

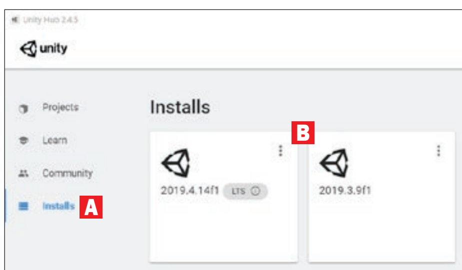
Unity Hub

Unity Hub wykorzystujemy do tego, by zainstalować odpowiednią wersję silnika Unity, ale też do tego by tworzyć nowe projekty, określając wersję, która zostanie do tego użyta. Jeśli przyjrzymy się Unity Hub dokładniej, zauważymy jeszcze więcej dodatkowych możliwości.

Instalacja wybranej wersji silnika

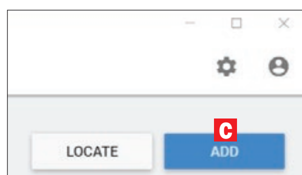
Okno Unity Hub jest podzielone na zakładki, pomiędzy którymi możemy przełączać się poprzez panel po lewej stronie okna programu.

1 By z poziomu Unity Hub dokonać instalacji wybranej wersji silnika, należy przejść do zakładki **Installs** A.



2 Jeżeli mamy już zainstalowane jakieś wersje silnika, zobaczymy je w oknie programu w formie odpowiadających im kafli B.

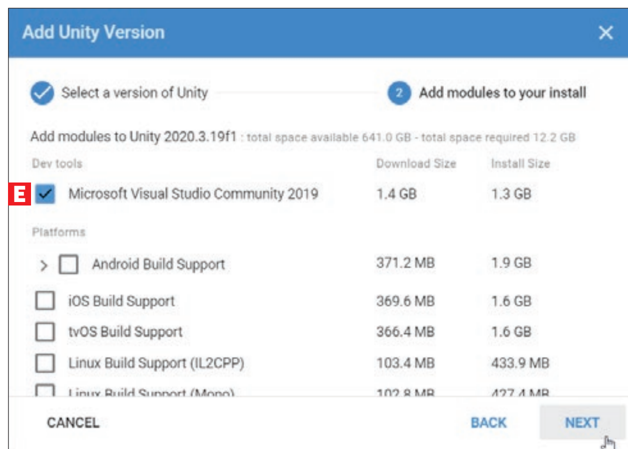
3 Jeśli nie mamy jeszcze potrzebnej wersji silnika Unity, zainstalujemy ją, klikając na widoczny w prawym górnym rogu okna przycisk **ADD** C.

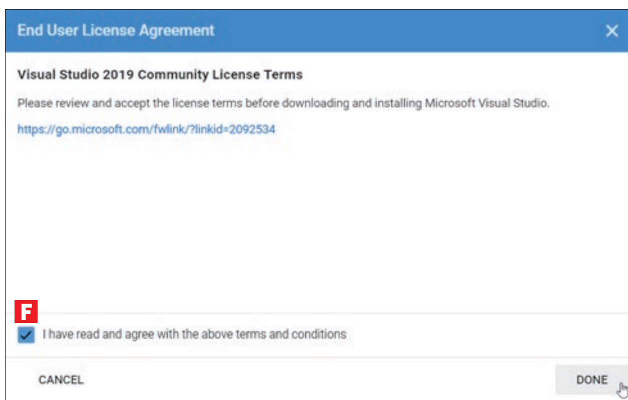


4 W nowym oknie możemy wybrać, jaką wersję Unity chcemy zainstalować. Część dostępnych wersji jest oznaczona jako **LTS** – są to wersje, dla których zapewnione będzie długoterminowe wsparcie i z których najlepiej korzystać. W tym momencie taką wersją jest **Unity 2020.3.19f1** D – jest ona oznaczona jako wersja rekomendowana i tej właśnie wersji dotyczyć będzie ta książka. Wiele elementów w różnych wersjach silnika jest takich samych, jednak by wykonywać wskazówki przedstawione na kolejnych stronach, warto zainstalować właśnie tę wersję – wtedy wszystkie kroki będą wyglądały tak samo. Po zaznaczeniu odpowiedniej wersji silnika przechodzimy dalej, klikając na **NEXT**.

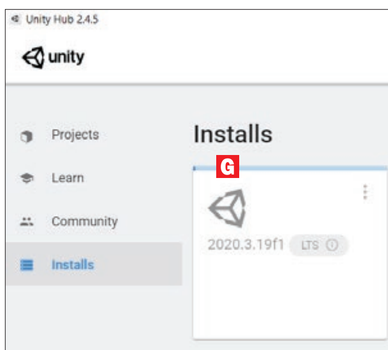


5 Teraz możemy wybrać dodatkowe moduły instalacji. Zaznaczamy narzędzie programistyczne **Microsoft Visual Studio Community 2019** **E**. Zapewni to dostęp do narzędzia pozwalającego między innymi na edycję kodów źródłowych. By móc z niego korzystać, przechodzimy dalej i akceptujemy warunki licencji, zaznaczając pole u dołu okna **F**. Klikamy na **DONE**, by zakończyć konfigurację i przejść do instalacji.





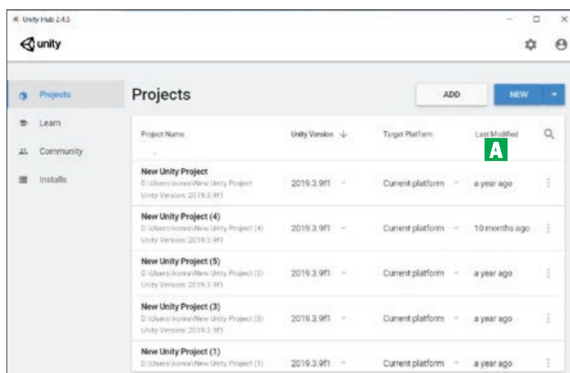
6 Teraz w zakładce **Installs** możemy obserwować postępy instalacji – są one widoczne na pasku nad nowym kaflem **G** instalowanej wersji silnika.



Tworzenie i uruchamianie projektów

Po zakończeniu instalacji wybranej wersji silnika możemy tworzyć nowe projekty.

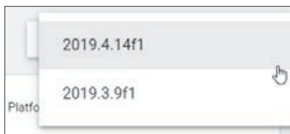
1 Przechodzimy do zakładki **Projects**. Kiedy będziemy już mieli własne projekty, to właśnie na tej zakładce zobaczymy ich zestawienie – nie tylko nazwy projektów, ale też wersje Unity wykorzystane do ich stworzenia, docelową platformę, na której mają działać, a także informacje o ostatniej modyfikacji każdego z nich (w kolumnie **Last Modified** **A**).



2 By stworzyć pierwszy nowy projekt, należy kliknąć na **NEW** (a jeśli mamy zainstalowanych kilka wersji Unity, klikamy na strzałkę **B** przy tym przycisku – zobaczymy



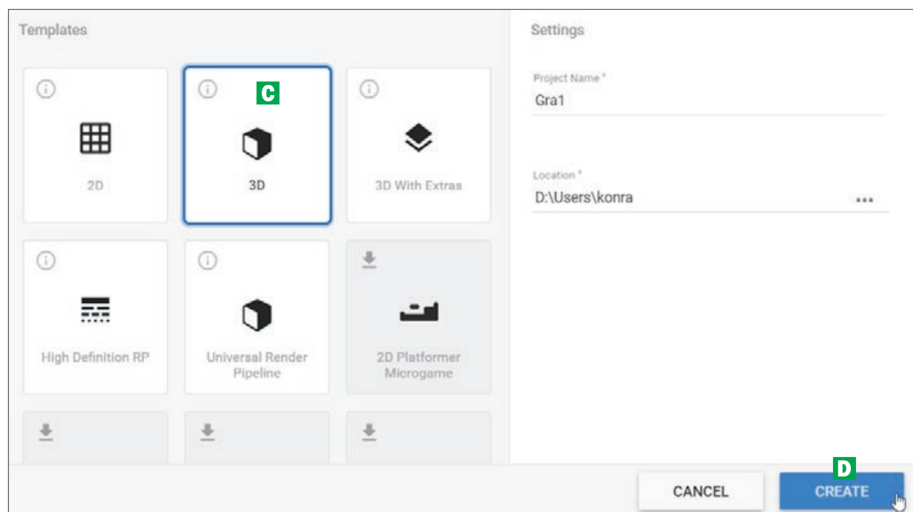
wtedy listę wersji silnika, które możemy wykorzystać do tworzenia nowego projektu).



3 By stworzyć nowy projekt 3D, zaznaczamy odpowiedni szablon – kafel **3D** **C** (w kolejnych wskazówkach będziemy zajmować się właśnie grami z grafiką trójwymiarową).

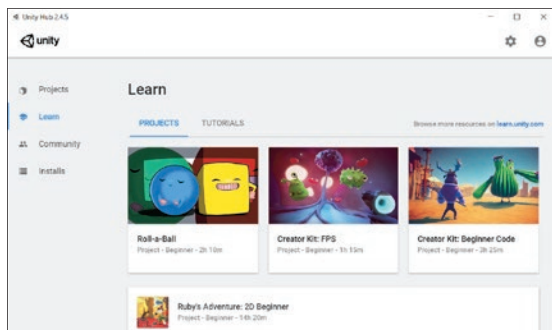
4 Uzupełniamy pola po prawej stronie okna, podając nazwę projektu (**Project Name**) i jego lokalizację na dysku. Następnie możemy utworzyć projekt, klikając na przycisk **CREATE** **D**.

5 Po kliknięciu na niego zobaczymy już uruchomione narzędzie Unity z utworzonym projektem.



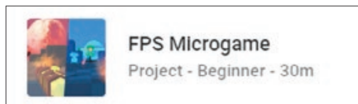
Gotowe projekty

Wiemy już, jak utworzyć własny, pusty projekt, jednak możliwości narzędzia łatwiej jest poznać, gdy w projekcie znajduje się już jakaś zawartość. Twórcy Unity przygotowali zestaw takich gotowych projektów, które można wykorzystać do nauki. Są one dostępne z poziomu **Unity Hub** – w zakładce **Learn**.

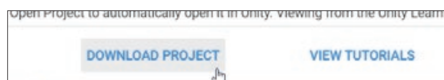


Na potrzeby omówienia zawartości edytora Unity wykorzystamy jeden z takich projektów.

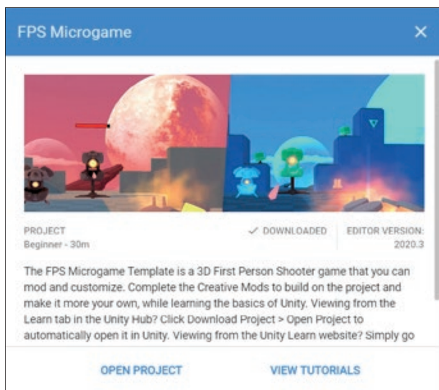
1 Znajdujemy na liście pozycję **FPS Microgame** i klikamy na nią, by otworzyć okno opisu projektu.



2 W nowym oknie wybieramy **DOWNLOAD PROJECT**, by pobrać zasoby projektu.



3 Gdy projekt jest już pobrany, możemy wybrać opcję **OPEN PROJECT**, by uruchomić Unity z wybranym projektem.



Unity – okno edytora

Gdy otworzymy Unity właśnie z takim gotowym projektem, na pierwszy rzut oka nie zobaczymy jego zawartości, edytor wygląda tak samo, jak wyglądałby z pustym projektem. Poznajmy elementy okna programu, aby dowiedzieć się, gdzie szukać zawartości importowanego projektu.

A To **obszar roboczy**. Zawiera widok aktualnie tworzonej sceny. A czym jest scena? Może reprezentować jeden z poziomów naszej gry bądź cały świat gry. Zależnie od obranego podejścia gra może być zbudowana z kilku scen lub też osadzona w obrębie jednej sceny.

B **Zawartość sceny** jest opisana w panelu po lewej stronie. Każdy dodany do sceny obiekt znajdzie się właśnie w drzewie obiektów w tym panelu.

C **Panel wybranego obiektu**. Przedstawia właściwości obiektu, który został zaznaczony na scenie lub w panelu po lewej stronie. Czym są właściwości? To na przykład położenie obiektu na mapie, jego wymiary, a także inne właściwości zależne od tego, z jakich komponentów jest zbudowany obiekt. Takie komponenty to ważny element silnika Unity. Dodając do obiektu różnego rodzaju predefiniowane komponenty, nadajemy obiektom kolejne zestawy zachowań. Takim komponentem może być na przykład siatka kolizji, dzięki której w grze możliwe jest wykrywanie kolizji pomiędzy obiektami. Jeśli mowa o kolizji – jest to bardzo ważny element budowania świata. Wykrycie kolizji między obiektem, który ma znajdować się na mapie, a powierzchnią tej mapy jest konieczne choćby po to, by obiekt nie spadł w przestrzeń pod mapą.

D **Zasoby projektu**, nazywane też assetami. Tu możemy zapoznać się z całą zawartością projektu. Jest ona podzielona na foldery, w których znajdują się jej poszczególne elementy. Zasobem projektu mogą być też sceny, z jakich zbudowano grę – tak jest w przypadku projektu, któremu przyglądamy się w tym rozdziale.

1 By zobaczyć, jak wygląda gra, której dotyczy dany projekt, należy w zasobach projektu otworzyć folder **FPS** **E**.

2 W nim znajduje się folder **Scenes** **F**, a w nim sceny budujące grę.

3 Sceny oznaczone są ikoną sześcianu. Wśród dostępnych scen znajdujemy i otwieramy **MainScene** **G**.

4 Po jej otwarciu w oknie podglądu sceny zobaczymy widok gry. Widać również, że panel po lewej stronie zapełnił się obiektami z gry **H**.

Menu jako scena

Jeśli dokładniej przyjrzymy się scenom zawartym w projekcie, zobaczymy, że znajduje się tam scena o nazwie **IntroMenu** **I**. Jest to scena pełniąca rolę menu projektu. Takie sceny są ważnym elementem wielu projektów. A jak stworzyć własne menu gry? Tego dowiemy się z kolejnego rozdziału.

3 Obsługa edytora Unity

Z tego rozdziału dowiemy się, jak obsłużyć edytor Unity. Nauczymy się tworzyć obiekty na scenie i przypisywać im pierwsze działania. Zobaczymy też, jak można stworzyć menu i obsługiwać dźwięki. To wiedza, która przyda się nam w pracy nad każdym projektem

Menu jest jednym z podstawowych elementów, jakie pojawiają się w grach. Jeśli chcemy tworzyć gry w Unity, warto rozpocząć właśnie od menu. Tworząc nasze pierwsze menu, poznamy tajniki obsługi tego rozbudowanego narzędzia.

SŁOWNICZEK POJĘĆ

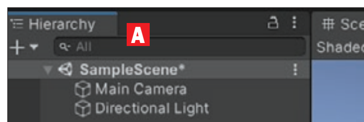
By lepiej zrozumieć to, co opisane zostało w tym i kolejnych rozdziałach, warto zapoznać się z podstawowymi pojęciami dotyczącymi struktury projektu w Unity.

POJĘCIE	OPIS
Obiekt	Obiektami nazywamy wszystkie elementy znajdujące się na scenie. Domyślnie po utworzeniu sceny znajdują się na niej dwa obiekty. Jeden to **Main Camera**, czyli obiekt decydujący o tym, który obszar sceny jest widoczny w oknie gry. A drugi to **Directional Ligh**t, czyli oświetlenie, dzięki któremu możliwe jest między innymi wyświetlanie cieni rzucanych przez inne obiekty.
Zasoby projektu	Każdy projekt tworzony w Unity składa się z wielu plików. Pliki te przechowywane są w jednym folderze. Jego zawartość nazywamy zasobami projektu. Z poziomu okna edytora Unity możemy przeglądać zawartość zasobów projektu. Zasoby te są inaczej nazywane **Assetami**.
Właściwości obiektu	Jest to zbiór cech charakterystycznych dla obiektów sceny. Takim cechom możemy przypisywać różne wartości, co ma wpływ na wygląd i zachowanie obiektów sceny. Edycję właściwości przeprowadzamy z poziomu panelu **Inspector**.
Scena	Scena to pojedynczy obszar gry, który może się załadować niezależnie od reszty. Scenami mogą być kolejne poziomy rozgrywki, panel ustawień, a także menu gry. Tworzone przez nas projekty będą mogły składać się z kilku scen. Jedną z nich może być menu główne gry.
Skrypty	Skrypty to inaczej kody napisane w wykorzystywanym w Unity języku programowania C#. Są one tym elementem tworzonych projektów, który zapewnia nam największą personalizację osiąganych efektów. Ich tworzenie wymaga jednak znajomości języka programowania i dostępnych w silniku Unity funkcji.

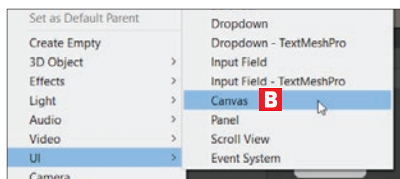
Scena jako podstawowy element projektu

Zaczynamy od stworzenia nowego projektu (patrz rozdział **2**, strona 11). Automatycznie otworzy się pusta scena. To na niej będziemy tworzyć menu. Do sceny należy zatem dodać obiekty, tak by w efekcie pojawiły się na niej przyciski – jeden do uruchomienia właściwej rozgrywki i drugi służący do jej zakończenia.

1 By dodać dowolny obiekt do sceny, należy kliknąć prawym przyciskiem myszy na panel nazywany **Hierarchy** **A** po lewej stronie, wyświetlający zawartość sceny. Dzię-

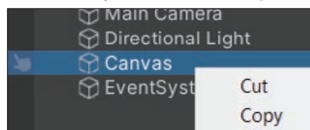


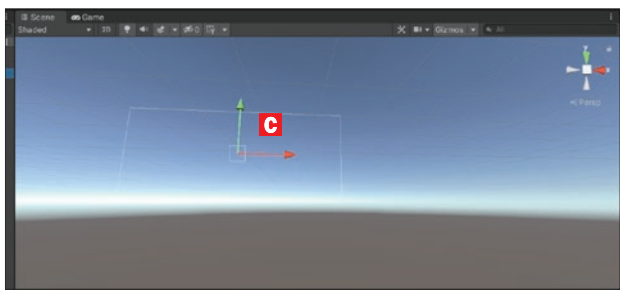
ki temu zostanie wyświetlone menu kontekstowe, którego dolna część to zestawienie kategorii nowych obiektów, jakie możemy dodać do projektu. Kategorią, w której znajdują się obiekty, jakie wykorzystamy do budowy menu, jest **UI**, i to ją należy rozwinąć, by odnaleźć pierwszy element budujący nasze menu, czyli **Canvas** **B**. Będzie on pełnił rolę płaszczyzny, na której podczas dalszej pracy umieścimy przyciski menu.



2 Obiekt powinien być widoczny także na scenie w formie białego prostokąta z zaznaczonymi na jego środku strzałkami **C**.

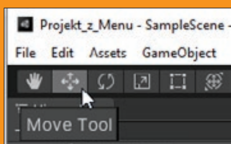
3 Menu powinno mieć tło. Ponownie należy dodać nowy obiekt, wywołując menu kontekstowe, tym razem jednak klikamy prawym przyciskiem myszy nie na wolny obszar panelu, ale na obiekt **Canvas**.



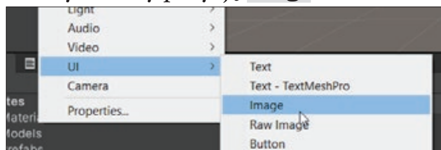


PORUSZANIE SIĘ PO SCENIE

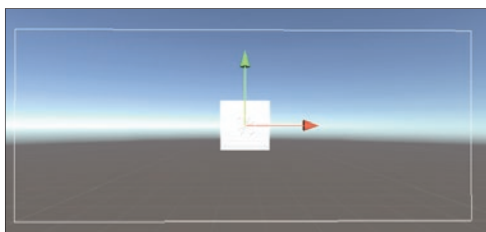
To, co widać w zakładce **Scene**, to tylko domyślny widok modelowanej sceny. Podgląd sceny możemy przesuwać. Sposób poruszania się po podglądzie sceny określamy za pomocą przycisków na pasku górnym pod menu głównym. W wypadku opcji **Move Tool** po scenie przemieszczamy się, wciskając kółko myszy i przesuwając kursor po podglądzie. Widok można również obracać, przesuwając kursor po scenie z wciśniętym prawym przyciskiem myszy. Obracając kółko myszy, można z kolei przybliżać i oddalać podgląd sceny. Niezależnie od wybranego sposobu poruszania się po scenie, jeśli chcemy na podglądzie zobaczyć konkretny obiekt, klikamy dwukrotnie na ten obiekt w panelu **Hierarchy** po lewej stronie okna edytora.

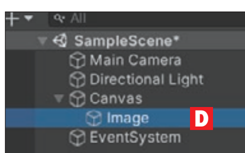


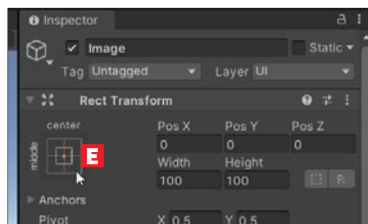
4 W menu kontekstowym z kategorii **UI** wybieramy pozycję **Image**.



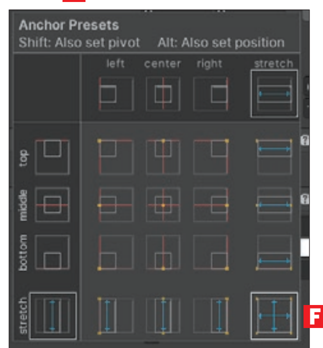
5 Po jego dodaniu na naszym menu pojawi się niewielki kwadrat. Należy go powiększyć w taki sposób, by wypełnił cały obszar menu, stanowiący jego tło. Można to zrobić, edytując jego właściwości na panelu **Inspector** po prawej stronie **D**.





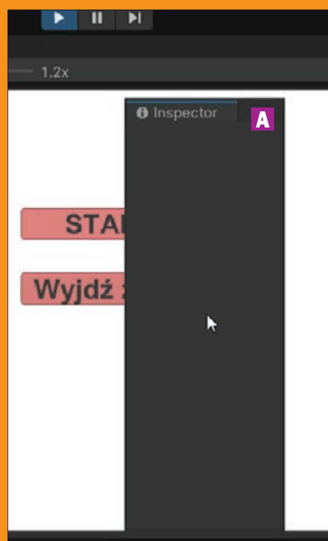


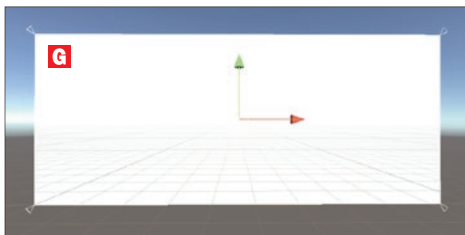
6 Klikamy na kwadrat znajdujący się w grupie **Rect Transform** **E**, by zobaczyć pełną listę opcji do wyboru. Z przedstawionej listy należy wybrać opcję rozciągnięcia, która jest widoczna przy wciśniętym klawiszu alt i znajduje się w prawym dolnym rogu zestawienia **F**.



ROZMIESZCZENIE PANELI W OKNIE EDYTORA

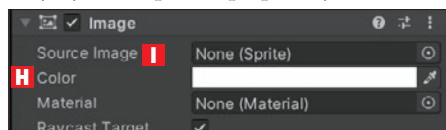
W treści tego rozdziału bardzo często padają określenia takie jak „panel po lewej stronie" czy „panel po prawej stronie", które odnoszą się do poszczególnych sekcji okna edytora Unity. Określenia te uwzględniają pierwotne, domyślne położenie każdego z paneli w oknie edytora Unity. Nie oznacza to jednak, że panele te zawsze muszą znajdować się w takich miejscach. Edytor Unity pozwala nam na dopasowanie rozmieszczenia poszczególnych paneli w zależności od preferencji użytkownika. By zmienić położenie wybranego panelu, klikamy prawym przyciskiem myszy na nazwę panelu i przesuwamy kursor. Panel zostanie odczepiony od aktualnej lokalizacji. Przemieszczając taki odczepiony panel **A** w inne miejsce, można przypisać mu nową lokalizację w oknie edytora.



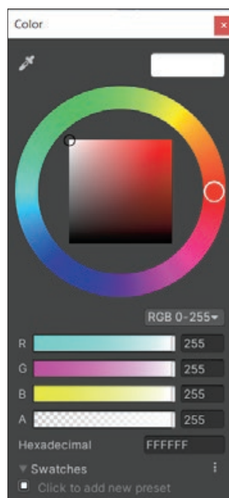


7 Obiekt **Image** będzie wypełniał już cały obszar **G** zajmowany przez **Canvas**.

8 Tło domyślnie jest koloru białego. Jeśli chcemy, by zmieniło ono kolor, należy zmienić właściwość **Color** **H**, która również znajduje się w panelu po prawej stronie.



9 Kliknięcie na pasek koloru spowoduje otwarcie nowego okna z paletą barw. Z jego poziomu możliwe jest określenie dokładnego koloru tła.

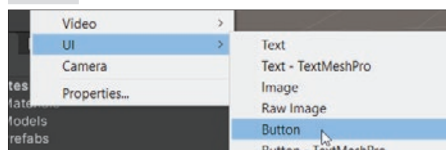


10 Tło nie musi być jednokolorowe. Jeśli chcemy, możemy ustawić jako tło menu własną grafikę. Najpierw należy jednak plik graficzny umieścić w folderze, w którym zlokalizowano projekt. Gdy plik będzie już w lokalizacji projektu, będzie można go wybrać z poziomu opcji **Source Image** **I**, tuż nad polem wyboru koloru.

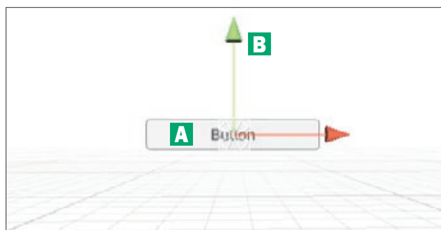
Przyciski menu

Gdy mamy już utworzone tło dla naszego menu, możemy zająć się tworzeniem jego właściwej treści. A tą właściwą treścią powinny być przyciski, jakie na nim umieścimy. Będziemy dodawać je podobnie, jak dodaliśmy do naszego menu tło.

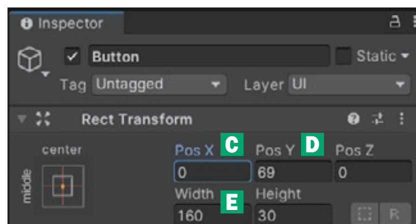
1 W panelu **Hierarchy** po lewej stronie klikamy prawym przyciskiem myszy na na obiekt **Canvas** i z menu kontekstowego, z kategorii **UI** wybieramy tym razem obiekt **Button**.



2 Po jego wybraniu niesformatowany przycisk **A** zostanie umieszczony w centralnym punkcie naszego menu.



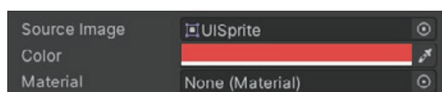
3 Jeśli chcemy zmienić położenie przycisku na menu, możemy to zrobić, klikając na strzałki widoczne na przycisku **B** lub korzystając z panelu po prawej stronie. Tam w grupie **Rect Transform** możemy edytować wartości pól **Pos X** **C**, które określa położenie przycisku w poziomie, oraz **Pos Y** **D**, które określa położenie przycisku w pionie.



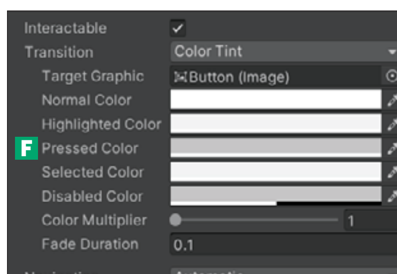
Jeśli obiekt ma w tych dwóch polach współrzędnych wartości 0, znajduje się na środku.

4 W tej samej grupie właściwości widzimy też pola **Width** i **Height** **E** – określają one odpowiednio szerokość i wysokość przycisku.

5 Również kolor przycisku może zostać zmieniony. Bazową barwę możemy określić w polu **Color**. Przycisk ma jednak różne



stany – na przykład może być wciśnięty lub nie. Jego wygląd w zależności od stanu, w jakim się znajduje, ulega zmianom. To też możemy doprecyzować, korzystając z zestawu właściwości, które znajdują się w tym samym



panelu, jednak nieco niżej. Zmiana wartości dla właściwości podpisanej jako **Pressed Color** **F** pozwala na modyfikację wyglądu przycisku, kiedy jest on wciśnięty. Wygląd przycisku zmienia się w sposób płynny, jego ostateczny kolor nie jest odzwierciedleniem wartości właściwości.

Zmiana napisu na przycisku

Na dodanym przez nas przycisku wciąż widoczny jest pierwotny napis **Button**.

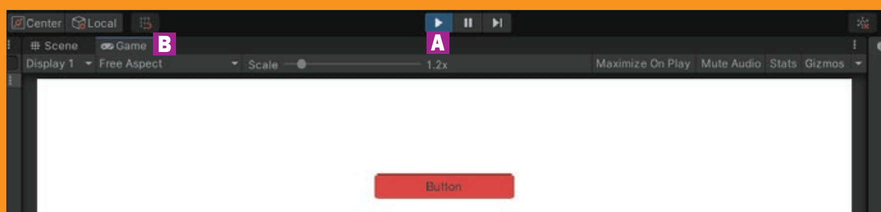
TRYB TESTOWANIA SCENY

Część zmian w wyglądzie przycisku, jakie wprowadzamy, jest widoczna od razu na podglądzie sceny. Widzimy jednak tylko te zmiany, które dotyczą wyglądu bazowego. A co ze zmianami, które powinny być widoczne dopiero w momencie zmiany stanu, w jakim znajduje się przycisk, a konkretnie po kliknięciu na niego? Możemy je zobaczyć dopiero po uruchomieniu całego projektu. Projekt możemy na bieżąco testować.

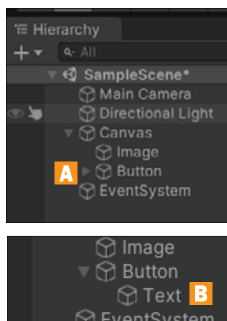
1 W górnej części okna edytora Unity znajduje się zestaw przycisków, z których pierwszy **A** służy do uruchamiania podglądu gry. Po kliknięciu na niego zostajemy przełączeni do zakładki **Game** **B**, w której znajduje się właśnie podgląd gry. Gdy mamy otwartą tę zakładkę, możemy kliknąć na widoczny na menu przycisk i zobaczyć, jak zmienia się jego wygląd w momencie klikania na niego.

2 Obserwując uruchomiony podgląd, wciąż możemy zmieniać właściwości na panelu po prawej stronie.

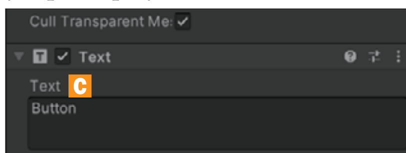
Warto wiedzieć: wprowadzane przez nas zmiany będą widoczne na podglądzie, jednak po zakończeniu obserwacji podglądu (czyli po ponownym kliknięciu na przycisk, którym uruchomiliśmy podgląd) zmiany wprowadzone w tym czasie zostaną cofnięte.



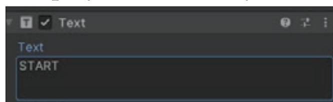
1 Żeby zmienić napis na przycisku, musimy wiedzieć, jak jest on zbudowany. Na panelu **Hierarchy** po lewej stronie, przy naszym przycisku podpisanym na strukturze obiektów jako **Button**, znajduje się niewielka strzałka **A**. Oznacza to, że tak naprawdę przycisk może zawierać w sobie jeszcze jakieś inne obiekty. Klikając na nią, zobaczymy, z czego składa się przycisk dodany przez nas do menu. Pokazuje się obiekt **Text** **B**.



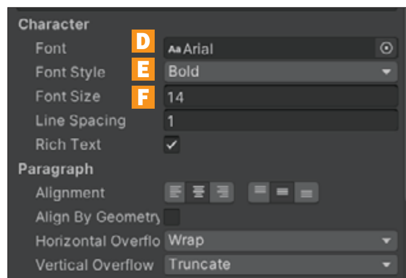
2 Zaznaczając ten obiekt na panelu **Hierarchy**, możemy wyświetlić właściwości samego tekstu na panelu po drugiej stronie okna edytora. To właśnie w tym zbiorze właściwości znajduje się pole **Text** **C**. Widzimy w nim napis **Button**. Zmieniając go, zmienimy napis na przycisku.



3 Pierwszy przycisk w menu będzie służyć do uruchamiania właściwej gry, a napis na przycisku powinien wskazywać na jego działanie. Może to być na przykład **START**. Działanie przycisku będziemy mogli zaprogramować dopiero wtedy, gdy będziemy mieć przygotowaną scenę, która uruchomi się wraz z kliknięciem na niego. W zależności od tego, do której z gier będzie to menu i w której go umieścimy, uruchamiana będzie inna scena.



4 Treść napisu na obiekcie tekstowym nie jest jedyną rzeczą, jaką możemy zmodyfikować z poziomu panelu właściwości tego obiektu. Możemy tu też zmienić na przykład rodzaj użytej czcionki – zrobimy to w polu podpisanym jako **Font** **D**, gdzie domyślnie użyta czcionka to **Arial**.



5 W polu podpisanym jako **Font Style** możemy nadać czcionce takie właściwości, jak pogrubienie (**Bold** **E**), pochylenie czy podkreślenie.

6 Ważną właściwością jest tu też **Font Size** **F**. Liczba wpisana w tym polu odpowiada za wielkość czcionki.
Co istotne, wielkość czcionki nie zmienia się wraz z modyfikacją wymiarów przycisku. Jeśli zechcemy mieć większe przyciski w menu, rozmiar tekstu pozostanie niezmieniony. Zatem by zachować czytelność tworzonego menu, rozmiar należy modyfikować ręcznie właśnie w tym menu.

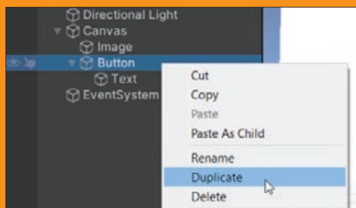
Dodawanie nowych przycisków

Przycisk do rozpoczynania gry to jeszcze nieco zbyt mało, aby móc mówić o pełnowartościowym menu gry. Na scenie powinny pojawić się zatem kolejne przyciski. Można byłoby dodać je, wykonując kroki opisane wcześniej w tym rozdziale, począwszy od dodania obiektu **Button** dla istniejącego już obiektu **Canvas**. Za każdym razem otrzymalibyśmy na początku niesformatowany przycisk ze standardowymi kolorami, wymiarami i tekstem, które należałoby zmienić na wzór pozostałych przycisków. Lepszym rozwiązaniem jest duplikowanie już istniejących obiektów. Stwórzmy teraz drugi przycisk menu – z napisem **Wyjdź z gry**.

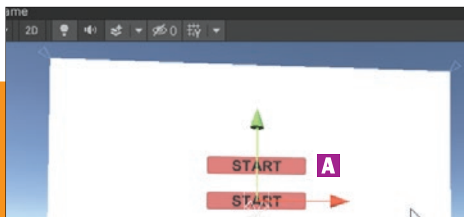
DUPLIKOWANIE OBIEKTÓW

1 Klikamy prawym przyciskiem myszy na obiekt, który chcemy zduplikować, w panelu obiektów **Hierarchy** po lewej stronie okna edytora.

2 Z wyświetlonego w ten sposób menu kontekstowego wybieramy opcję **Duplicate**. Jej wybór sprawi, że w zestawieniu

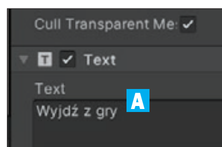


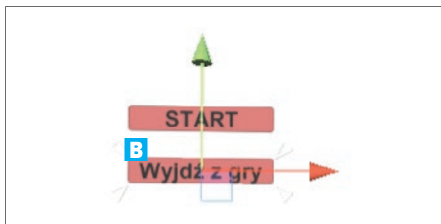
obiektów pojawi się duplikat istniejącego już obiektu. W przypadku duplikowania przycisku byłby to **Button (1)**.



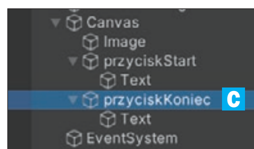
3 Nowy obiekt jest na scenie, jednak na pierwszy rzut oka jest niewidoczny. To dlatego, że gdy tworzymy duplikat, ma on skopiowane z oryginału wszystkie właściwości – a do nich należy także lokalizacja na scenie. Zatem mamy dwa przyciski, które znajdują się w tym samym miejscu. Żeby móc zmienić lokalizację nowego przycisku, powinien on być zaznaczony na panelu po lewej stronie, wtedy strzałki widoczne na podglądzie sceny pozwolą na przesunięcie go w trybie ręcznym. Również właściwości widoczne na panelu po prawej stronie będą dotyczyły aktualnie zaznaczonego obiektu. Po przesunięciu powinny być już widoczne oba obiekty **A**.

1 Nowy napis na przycisku ustawimy z poziomu właściwości **Text** obiektu **Text** należącego do nowego przycisku **A**. Po wpisaniu tekstu w to pole odpowiedni napis pojawi się na nowym przycisku **B**.



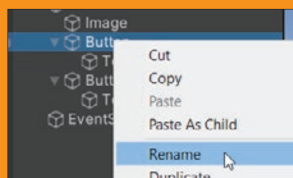


2 Napisy na przyciskach już zmienialiśmy, ale nazwy obiektów widoczne na panelu po lewej stronie wciąż są niezmienione. Jeśli w naszym menu pojawi się więcej przycisków, trudno nam będzie rozpoznać, która nazwa odnosi się do jakiego przycisku – w panelu ze spisem obiektów sceny wszystkie będą się nazywały Button. Można temu zaradzić, ustalając nowe nazwy dla naszych przycisków – takie, które pozwolą je zidentyfikować, na przykład **przyciskStart** i **przyciskKoniec** **C**.



ZMIANA NAZWY OBIEKTU

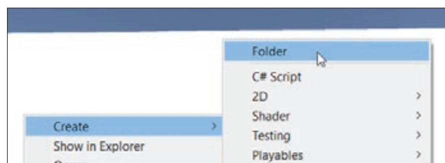
Nową nazwę obiektowi możemy nadać, klikając na niego prawym przyciskiem myszy i z menu kontekstowego wybierając **Rename**.



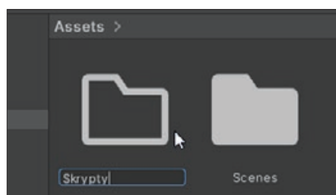
Przypisanie działania do przycisku

Przyciski dodane do naszego menu nie mają jeszcze ustalonego działania. O ile zaprogramowanie przycisku startującego grę będzie możliwe dopiero wtedy, kiedy w projekcie znajdzie się scena, którą ma uruchamiać kliknięcie na ten przycisk, o tyle zaprogramowanie przycisku kończącego grę możliwe jest na tym etapie pracy. Niezależnie jednak od tego, jakie jest przeznaczenie przycisku, część czynności, jakie należy wykonać, by mógł działać, jest niezmienna.

1 Działanie przycisku powinien opisywać skrypt, który będziemy musieli napisać. Ponieważ w projekcie gry może być wiele skryptów opisujących działanie różnych elementów gry, warto w tym celu przygotować w zasobach projektu odpowiednie miejsce do przechowywania skryptów. Dlatego zanim przystąpimy do pisania skryptu, w zasobach projektu należy utworzyć folder do magazynowania skryptów. Klikamy prawym przyciskiem myszy na panel zasobów projektu u dołu okna edytora, by otworzyć menu kontekstowe. Rozwijamy w nim sekcję **Create** i wybieramy opcję **Folder**.



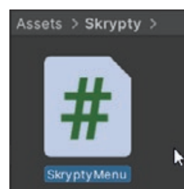
2 Nowemu folderowi w zasobach projektu nadajemy nazwę **Skrypty**.



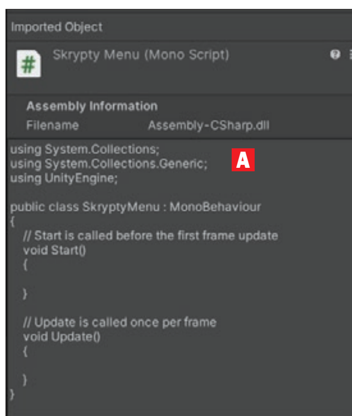
3 Otwieramy utworzony folder. W nim trzeba będzie utworzyć skrypt, w którym opisane zostanie działanie przycisków. Tworzenie skryptu można przeprowadzić z poziomu menu kontekstowego. Po jego rozwinięciu, podobnie jak było to w przypadku tworzenia folderu, znajdujemy opcję **Create**, a ze znajdujących się w niej pozycji wybieramy następnie **C# Script**.



4 W ten sposób utworzony zostanie plik ze skryptem, docelowo powinny w nim się znaleźć skrypty obsługujące menu, dlatego dobrą nazwą dla pliku będzie **SkryptyMenu**.

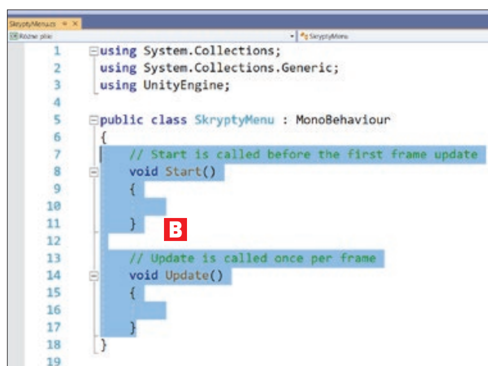


5 Gdy plik jest zaznaczony w zasobach projektu, panel po prawej stronie wyświetla jego właściwości, w tym przypadku wyświetla on treść skryptu **A**. Nie mamy możliwości edycji tej treści z poziomu panelu właściwości.



6 By edytować skrypt, należy go otworzyć dwukrotnym kliknięciem, jak w folderze otwartym w oknie Eksploratora. Plik zostanie otwarty w programie **MS Visual Studio**, który powinien być domyślnym edytorem

kodów dla plików z rozszerzeniem **.cs**. W treści skryptu znajdują się dwie funkcje, **Start** i **Update**, o których działaniu dowiemy się z kolejnych rozdziałów tej książki. Na razie funkcje te nie są nam potrzebne, dlatego zarówno jedną, jak i drugą można usunąć **B**.



7 Zamiast nich w skrypcie powinna się znaleźć funkcja, która docelowo będzie odpowiedzialna za zamykanie gry. By utworzyć funkcję o nazwie **Zamykanie**, zapisujemy **public void Zamykanie()**.

```
public class SkryptyMenu : MonoBehaviour
{
    public void Zamykanie()
}
```

8 W skrypcie nie ma jeszcze miejsca, w którym będzie można zapisać treść utworzonej funkcji. Taką treść zapisuje się w nawiasie klamrowym, zatem dalej należy utworzyć taki nawias **C**.

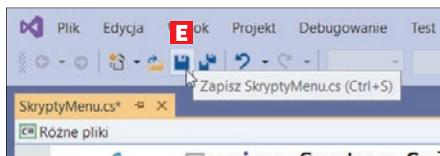
```
public class SkryptyMenu : MonoBehaviour
{
    public void Zamykanie()
    { C

    }
}
```

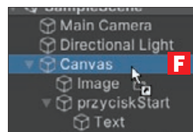
9 W jego wnętrzu można zapisać polecenia, które mają się wykonać, gdy przycisk zamykający grę będzie „kliknięty". Zamknięcie aplikacji zapiszemy jako **Application.Quit(); D**.

```
public void Zamykanie()
{
    Application.Quit(); D
}
```

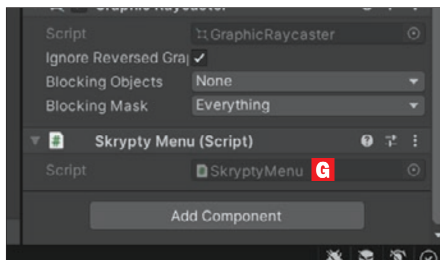
10 Po wpisaniu odpowiedniej treści do skryptu należy zapisać **E** wszystkie wprowadzone do niego zmiany, by były one uwzględnione przez edytor Unity.



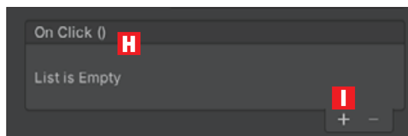
11 Plik ze skryptem powinien być połączony z którymś z obiektów na scenie, by możliwe było wykonywanie zawartych w nim funkcji. Jak dokonać takiego połączenia? Wystarczy przeciągnąć plik z panelu z zasobami projektu na obiekt na panelu po lewej stronie, do którego chcemy przyłączyć skrypt. Tym obiektem może być **Canvas F**, ponieważ to on pełni rolę podstawy menu i jest obiektem nadrzędnym dla przycisków, których działanie będzie docelowo opisane funkcjami zawartymi w tym pliku ze skryptem.



12 Po przeciągnięciu skryptu na obiekt zaznaczamy obiekt **Canvas** w zestawieniu obiektów i na panelu właściwości po prawej stronie sprawdzamy, czy jest tam wyświetlana nazwa utworzonego i połączonego skryptu **G**.

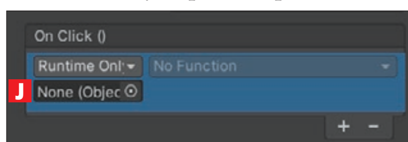


13 Gdy skrypt jest już połączony z odpowiednim obiektem, można przejść do przycisku do zamykania gry i z poziomu jego właściwości ustawić, aby funkcja **Zamykanie** z napisanego skryptu była wywoływana w momencie klikania na przycisk. A to, co ma się dziać w momencie klikania na przycisk, ustawiamy w sekcji **On Click ()** **H**.

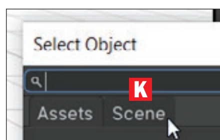


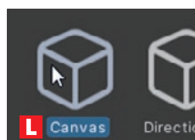
Jak widać, znajduje się tam teraz informacja **List is Empty**. Oznacza ona, że nie ma jeszcze wybranej żadnej akcji, która miałaby się wykonać po kliknięciu na przycisk. By jakakolwiek akcja się wykonała, klikamy na przycisk z plusem **I** w prawym dolnym rogu tej właściwości.

14 Zostanie w ten sposób utworzona nowa akcja do wykonania. W niej powinniśmy ustalić, jaka funkcja ma być w tej akcji wywołana. Jednak by wybrać funkcję, należy najpierw określić obiekt, do którego jest przypisany skrypt z napisaną przez nas funkcją do wykonania. By móc ten obiekt określić, klikamy na pole z napisem **None** **J**.

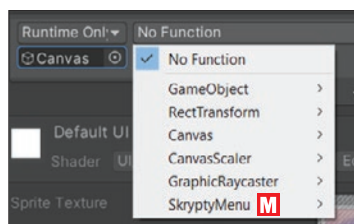


15 Zostanie otwarte nowe okno, z poziomu którego możemy wybrać element, dla którego mamy funkcję do wykonania. Okno jest podzielone na dwie zakładki, **Assets** i **Scene**. W drugiej z nich, czyli **Scene** **K**, znajdują się obiekty ze sceny. I właśnie w grupie obiektów ze sceny znajduje się podstawa naszego menu, czyli obiekt **Canvas** **L**, do którego to przypisaliśmy wcześniej skrypt z interesującą nas funkcją do zamykania gry.

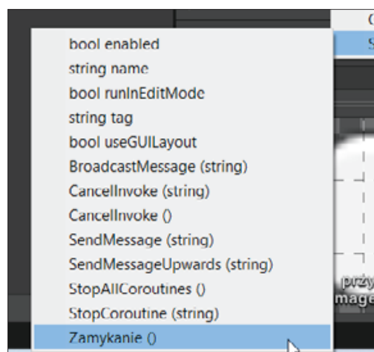




16 Po wybraniu obiektu możemy przejść do określenia funkcji. Robimy to w polu, w którym jest teraz napis **No Function** – po kliknięciu na ten napis zostaje wyświetlona lista opcji, wśród których znajduje się pozycja **SkryptyMenu** **M** odpowiadająca plikowi ze skryptem załączonemu przez nas do obiektu **Canvas**.



17 To właśnie w tej grupie opcji znajduje się poszukiwana przez nas funkcja do zamykania gry, czyli **Zamykanie()**, którą należy wybrać z listy.

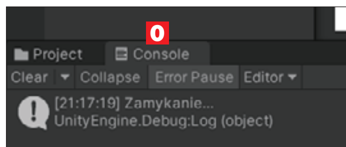


18 Przy uruchomieniu podglądu sceny nie przetestujemy, niestety, działania polecenia zamykającego aplikację, ponieważ działa ono jedynie na gotowym i wyeksportowanym projekcie. Możemy jednak przetestować, czy sama funkcja zamykająca jest prawidłowo wywoływana. Do funkcji **Zamykanie** moglibyśmy dopisać polecenie

Debug.Log("Zamykanie..."); N. Polecenie **Debug.Log** służy do wyświetlania tekstów w oknie konsoli i jest wykorzystywane do sprawdzania poprawnego działania elementów gry.

```
public void Zamykanie()
{
    Application.Quit();
    Debug.Log("Zamykanie...");N
}
```

19 Gdy skrypt jest zmodyfikowany, możemy już uruchomić podgląd sceny i kliknąć na przycisk służący do zamykania gry. Komunikat **Zamykanie...** powinien zostać wyświetlony w oknie konsoli. By dostać się do okna konsoli, należy przełączyć się na zakładkę **Console** O w panelu wyświetlającym zawartość projektu, u dołu okna edytora Unity.



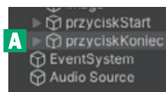
Dźwięk kliknięcia na przycisk

Unity pozwala nam także na obsługę dźwięków. W wielu aplikacjach obecny jest dźwięk kliknięcia na przycisk. Taki efekt możemy dodać także w tworzonym przez nas menu.

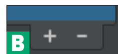
1 Za odtwarzanie dźwięków w obrębie sceny w Unity odpowiadają obiekty Audio Source. Taki obiekt należy utworzyć w naszej scenie z menu. Wybieramy do wstawienia obiekt **Audio Source**, znajdziemy go w kategorii **Audio**.



2 Dodany obiekt jest niewidoczny na scenie (choć jest widoczny w panelu A **Hierarchy**), jednak wykorzystując zawarte w nim mechanizmy, możemy odtwarzać dowolne dźwięki, jakie znajdują się w zasobach projektu. By odtworzyć dźwięk klikania na przycisk, należy przejść do sekcji **On Click()** we właściwościach przycisku.



3 Niezależnie od tego, czy lista akcji przypisanych do kliknięcia na przycisk jest pusta (jak w przypadku przycisku startującego grę), czy znajduje się tam już połączenie ze skryptem (jak w przypadku przycisku zamykającego grę), klikamy na plus B, by dodać nową akcję.

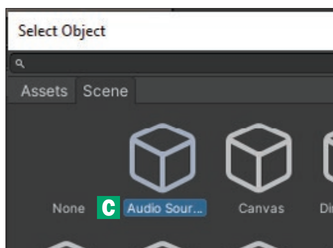


4 W polu, w którym jest napis **None (Object)**, należy z grupy obiektów sceny wybrać obiekt **Audio Source** C.

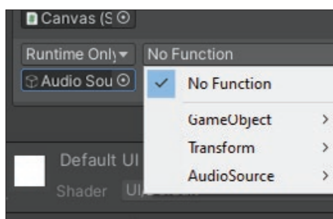
ZASOBY DŹWIĘKOWE

Jeśli potrzebujemy plików dźwiękowych, które chcemy wykorzystać w swoich produkcjach, możemy zapoznać się z zawartością portalu **opengameart.org**. W kategorii **Music** znajdziemy tam liczące tysiące pozycji zestawienie dźwięków do pobrania i wykorzystania w projekcie.

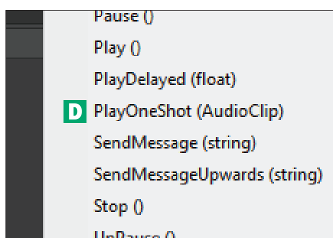




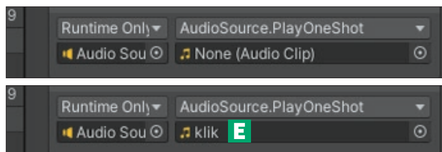
5 Rozwijając listę w polu **No Function**, zobaczymy, że mamy do wyboru grupę funkcji **AudioSource**.



6 Z listy funkcji należy wybrać **PlayOneShot (AudioClip)** D. Jest to funkcja wywołująca jednokrotne odtworzenie wybranego przez nas dźwięku.



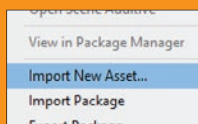
7 W polu akcji powinien pojawić się nowy slot, w którym wpisane jest **None (Audio Clip)**, co oznacza, że nie wybrano jeszcze dźwięku do odtworzenia przy klikaniu na przycisk. Dźwięk do slotu należy przeciągnąć z zasobów projektu. Po przeciągnięciu w slocie pojawi się nazwa E wybranego pliku dźwiękowego.



8 Gdy teraz uruchomimy podgląd sceny, przy klikaniu na przycisk będzie odtwarzany dźwięk.

SZYBKIE DODAWANIE PLIKÓW DO ZASOBÓW PROJEKTU

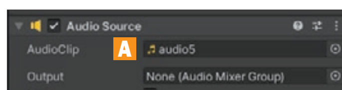
By szybko umieścić plik w zasobach projektu, nie musimy wyszukiwać folderu z projektem w Eksploratorze i wklejać tam potrzebnych nam zasobów. Możemy to zrobić bezpośrednio z poziomu edytora Unity – klikamy prawym przyciskiem myszy na panel zasobów projektu i z menu kontekstowe wybieramy **Import New Asset**.



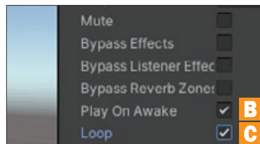
Muzyka odtwarzana w tle

Jeśli dysponujemy już obiektem **Audio Source**, można wykorzystać go nie tylko do odtwarzania dźwięków kliknięć na przyciski, ale również do odtwarzania muzyki w tle, gdy w naszej grze jest uruchomiona scena z menu. Cały mechanizm odtwarzania dźwięku ustawimy z poziomu właściwości tego obiektu.

1 Plik dźwiękowy, z którego muzyka ma być odtwarzana w tle, powinien znajdować się w zasobach projektu, tak abyśmy z tych zasobów mogli przeciągnąć go do slotu w polu **AudioClip** A.



2 Aktywna we właściwościach opcja **Play on Awake** **B** będzie oznaczała, że wybrany przez nas plik dźwiękowy będzie odtwarzany już w momencie załadowania obiektu na scenę, czyli w momencie uruchomienia.



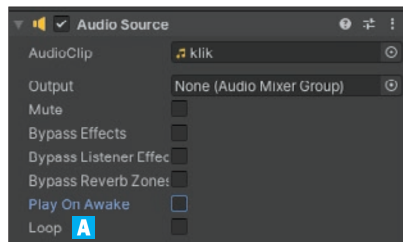
3 Jeśli zaznaczona zostanie opcja **Loop** **C**, odtwarzanie pliku dźwiękowego zostanie zapętlone. To dobre rozwiązanie, jeśli nie chcemy, by muzyka w tle przestała być odtwarzana.

Dźwięki w rozgrywce

Odtworzenie dźwięku w momencie kliknięcia na przycisk było łatwe do zaprogramowania i nie wymagało od nas wchodzenia w skrypty i poznawania odpowiednich poleceń dotyczących obsługi dźwięku. Stało się tak w głównej mierze dlatego, że przyciski, do których przypisywaliśmy dźwięk, mają w swoich właściwościach zdarzenie **On Click()**.
Może się jednak zdarzyć, że zechcemy odtwarzać dźwięki nie w momencie kliknięcia na przycisk, ale na przykład w chwili zdobycia przez gracza punktu w grze.
Taki moment jest zazwyczaj opisany odpowiednim skryptem. Polecenia odtwarzającego dźwięk można używać także w takim miejscu skryptu, w którym zostaje wychwycone zdobycie punktu i zwiększona liczba punktów.

1 Polecenie to wymaga jednak nie tego, by skorzystać z innego obiektu odtwarzającego dźwięki, ale tego, by dodać je do obiektu, którego skrypt wywoła odtwarzanie dźwięku komponentu – **Audio Source**.

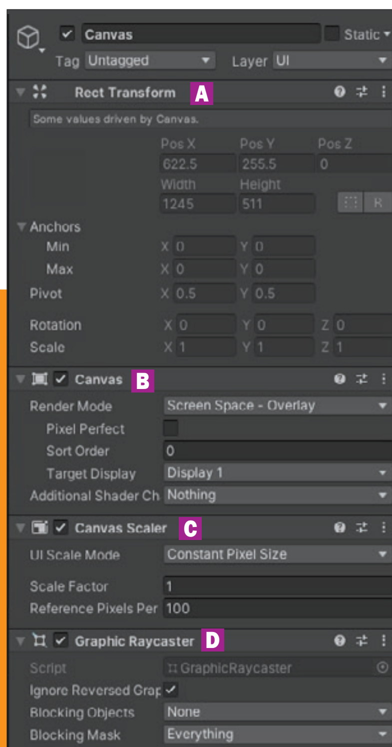
2 Gdy komponent jest dodany, dźwięk, który ma być odtworzony przez obiekt, powinien być przeciągnięty z zasobów projektu do slotu **Audio Clip** w komponencie obiektu. W tym samym komponencie opcje **Play on Awake** i **Loop** **A** nie powinny być zaznaczone. W przeciwieństwie do odtwarzania muzyki w tle, tu chcemy tylko wywołać jednorazowe odtworzenie dźwięku, które ma się rozpocząć w momencie wychwyconym i określonym przez skrypt, a nie w momencie pojawienia się obiektu na scenie.



3 Moment uruchomienia odtwarzania dźwięku zależy od konkretnej gry, w której ma być on odtworzony. Jednak polecenie, którym odtwarzamy dźwięk, jest zwykle takie samo i ma postać: **GetComponent<AudioSource>().PlayOneShot(GetComponent<AudioSource>().clip);**, gdzie **GetComponent<AudioSource>()** odnosi się do komponentu dodanego do obiektu, **PlayOneShot** jest nazwą funkcji, a **GetComponent<AudioSource>().clip** odczytuje pole **AudioClip** z właściwości komponentu. Gdybyśmy chcieli zastosować ten sposób odtwarzania dźwięku do kliknięcia na przycisk w naszym menu, komponent **Audio Source** powinien być dodany do tego samego obiek-

```
public void Zamykanie()
{
    Application.Quit();
    Debug.Log("Zamykanie..."); B
    GetComponent<AudioSource>().PlayOneShot(GetComponent<AudioSource>().clip);
}
```

tu, do którego jest przypięty skrypt, czyli do obiektu **Canvas**, a linia kodu zawierająca to polecenie powinna być wpisana do funkcji przypisanej do kliknięcia na przycisk. Dla odtwarzania dźwięku przy kliknięciu na przycisk zamykający grę powinna się ona znaleźć w funkcji **Zamykanie** B.



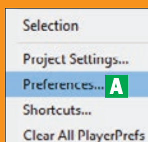
KOMPONENTY OBIEKTU

Zauważmy, że w panelu właściwości poszczególnych obiektów są podzielone na sekcje. I tak dla przykładu obiekt **Canvas** w swoich właściwościach ma sekcje **Rect Transform** A, **Canvas** B, **Canvas Scalar** C i **Graphic Raycaster** D. Takie pogrupowanie właściwości nie wynika z podziału ich na kategorie. Jest ono efektem tego, że obiekty zbudowane są z komponentów. Każdy obiekt można rozbudowywać poprzez dodawanie do niego nowych komponentów. Robimy to, klikając na przycisk **Add Component** u dołu panelu właściwości.

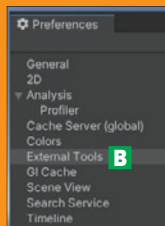
WYBÓR DOMYŚLNEGO EDYTORA KODÓW

Jeśli instalacja silnika Unity wykonana została zgodnie z instrukcją z tej książki, wraz z Unity powinno zostać zainstalowane środowisko programistyczne **Visual Studio 2019** i powinno być ono domyślnym programem, w jakim można edytować skrypty. Jeżeli mamy doświadczenie w tworzeniu kodów źródłowych i mamy ulubione programy do ich edycji, być może zechcemy użyć takiego programu również do edycji skryptów języka C# podczas realizacji opisanych projektów. Edytor Unity pozwala na określenie, w jakiej aplikacji mają być uruchamiane skrypty.

1 W menu górnym klikamy na pozycję **Edit**

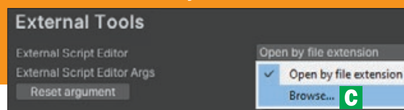


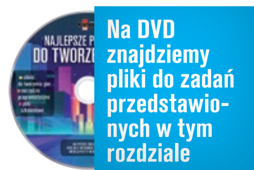
i z listy opcji wybieramy **Preferences** A.



2 W nowym oknie przechodzimy do zakładki wyboru narzędzi zewnętrznych, wybierając opcję **External Tools** B.

3 Rozwijamy pole znajdujące się przy pozycji **External Script Editor** i wybieramy opcję **Browse** C, która otwiera okno dialogowe. Z niego możemy wybrać aplikację, która ma otwierać pliki języka C# tworzone w Unity.





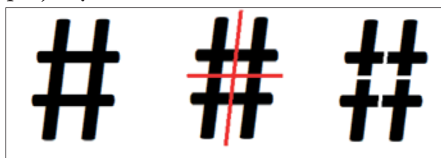
4 Język C#

C# to język programowania, który został wykorzystany w silniku gier Unity do tworzenia skryptów. Warto poznać jego podstawy, by sprawnie radzić sobie z programowaniem podczas tworzenia gier

Już przy okazji tworzenia menu pojawiła się konieczność skorzystania ze skryptów. Co prawda zrobiliśmy to bez konieczności posiadania specjalistycznej wiedzy, zanim jednak przejdziemy do tworzenia własnych gier, warto zapoznać się z językiem programowania, z jakiego przyjdzie nam korzystać, tworząc skrypty. Mowa tu o języku C#.

Nazwa i inne ciekawostki o języku C#

Jeśli znamy już podstawy programowania w języku C++, poznanie języka C# przyjdzie nam z łatwością, ponieważ w wielu elementach języki te są do siebie podobne, szczególnie jeśli chodzi o składnię podstawowych programistycznych konstrukcji, jak pętle czy instrukcja warunkowa.
Również nazwa języka C# nawiązuje do języka C++ i nie chodzi tu tylko o samą literę C, która jest nazwą jeszcze starszego od wymienionych języka programowania. C++ jest następcą języka C, do którego dodano nowe możliwości – stąd plusy w jego nazwie. Mówi się, że podobny zabieg zastosowano w nazwie języka C#, że tak naprawdę chodzi o C++++. Spójrzmy na znak # – jeśli podzielimy go na cztery części, zobaczymy cztery plusy. Ale to tylko ciekawostka na temat powstania nazwy języka. Tak naprawdę warto jednak poznać jego składnię i podstawowe pojawiające się w nim konstrukcje. A tego właśnie dowiemy się z dalszej części tego rozdziału. Wiedzę tę wykorzystamy potem w kolejnych rozdziałach, tworząc własne projekty.



Portal ideone.com

Portal **ideone.com** dostarcza nam narzędzie pozwalające na kompilację i debugowanie skryptów napisanych w ponad 60 językach programowania. Wśród tych języków jest też C#. Dlatego do poznania podstawowych elementów tego języka programowania wykorzystamy właśnie ten portal. Otwieramy zatem w przeglądarce internetowej stronę o adresie ideone.com **A**.

Wybór języka

Wiemy już, że portal, z którego mamy skorzystać, pozwala na programowanie w wielu językach. Chcąc cokolwiek napisać i sprawdzić działanie napisanych przez nas skryptów, powinniśmy zatem najpierw wybrać język programowania.

1 Pod polem tekstowym z zawartością skryptu znajdują się przyciski. Pierwszy z nich ma niewielką strzałkę zwróconą ku górze **B**, a jest na nim napisana nazwa języka programowania, który jest aktualnie wybrany i w którym jest napisany skrypt widoczny wyżej na stronie.

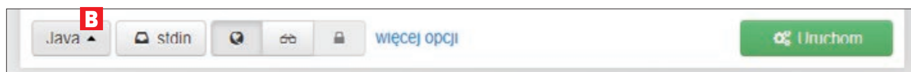
2 Kliknięcie na ten przycisk spowoduje wyświetlenie zestawienia dostępnych do wyboru języków programowania. Wśród nich znajduje się interesujący nas język – **C#** **C** – odnajdziemy go w pierwszej kolumnie, w grupie języków określanych jako popularne. Oprócz niego znajdują się tam takie języki, jak Java, C++, Python czy PHP.

3 Zmiana języka programowania spowoduje, że zmieni się też zawartość pola ze skryptem. Od teraz powinno ono już przedstawiać skrypt **D** zapisany w języku C#, którego to składnię będziemy omawiać w tym rozdziale.

```
using System;

public class Test
{
    public static void Main()
    {
        // your code goes here
    }
}
```

4 Jak widać, podobnie jak w przypadku skryptu tworzonego w edytorze Unity – tu również już na samym starcie mamy do dyspozycji pewien szkielet skryptu. Choć różnią się one między sobą, obydwa zawierają podstawowe elementy niezbędne do dalszej pracy.



KOMENTARZE

W szkielecie skryptu na stronie internetowej jest nawet wyznaczone miejsce, w którym powinniśmy rozpocząć pisanie własnych poleceń. To 7. linia kodu źródłowego. Strona, kolorując skrypt, pokolorowała ją na inny kolor – na zielono.

</> wprowadź kod źródłowy albo wstaw wzorzec lub przykład

```
1  using System;
2
3  public class Test
4  {
5      public static void Main()
6      {
7          // your code goes here
8      }
9  }
```

To dlatego, że ta linia kodu rozpoczyna się od dwóch ukośników. W języku C# w ten sposób oznaczamy komentarze. To specjalne fragmenty kodu, które nie są brane pod uwagę podczas jego wykonywania. W tym wypadku napis **your code goes here** oznacza, że „w tym miejscu powinien znaleźć się twój kod". Nie jest to żadne polecenie języka C#, a jedynie wskazówka dla nas – czyli użytkowników strony. Właśnie do zapisywania takich wskazówek możemy wykorzystywać komentarze. W ten sposób możemy podpisywać części kodu, abyśmy po dłuższym czasie mogli sobie łatwo przypomnieć, co zawierają.

Dyrektywa using

Przyglądając się skryptowi z portalu ideone.com i temu pochodzącemu z silnika Unity, widzimy, że oba zaczynają się od tego samego słowa. Jest to słowo **using**, w przypadku skryptu ze strony użyte w linii **using System;** **A**, a w przypadku kodu z Unity jest to **using System.Collections;** **B**.

A
```
1  using System;
```

B
```
1  using System.Collections;
```

Dyrektywa using umożliwia używanie typów zdefiniowanych w przestrzeni nazw bez określania w pełni przestrzeni nazw tego użytego typu. W podstawowej postaci dyrektywa importuje wszystkie typy z pojedynczej przestrzeni nazw. Najlepiej jednak mechanizm ten zaprezentować na przykładzie, edytując skrypt na stronie o adresie **ideone.com**.

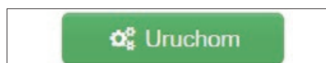
1 W miejscu komentarza wpisujemy linię **Console.WriteLine("Testowanie");** **C**. Jej użycie sprawia, że na wyjściu danych

```
using System;

public class Test
{
    public static void Main()
    {
        Console.WriteLine("Testowanie");
    }
}
```
C

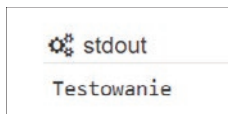
z programu powinno pojawić się to, co zapiszemy w cudzysłowie, czyli **Testowanie** – jest to polecenie podobne do **Debug.Log** znanego nam już z Unity.

2 By sprawdzić, czy zapisany skrypt działa prawidłowo, należy uruchomić go poprzez kliknięcie na przycisk **Uruchom** znajdujący się pod polem na skrypt.

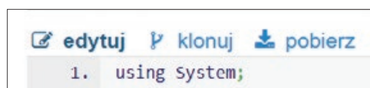


3 Po uruchomieniu nieco niżej pojawia się na stronie sekcja **stdout** zawierająca wyjście danych z programu. To właś-

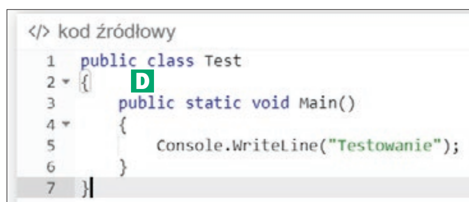
nie tam wypisane jest oczekiwane przez nas słowo – **Testowanie**.



4 Skrypt działa prawidłowo między innymi dlatego, że użyto dyrektywy **using System**. Co by się stało, gdyby zabrakło jej w skrypcie? Możemy to sprawdzić. Należy w tym celu edytować skrypt. Będzie to możliwe, gdy klikniemy na znajdujący się nad polem ze skryptem przycisk **edytuj**.



5 Gdy będziemy już w trybie edycji skryptu, kasujemy linię **using System;**, tak by kod źródłowy rozpoczynał się od **public class Test** **D**.

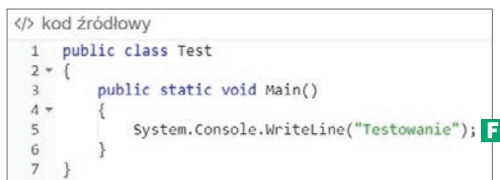


6 By zatwierdzić wprowadzone w treści skryptu zmiany, klikamy na przycisk **wyślij** widoczny pod polem ze skryptem.



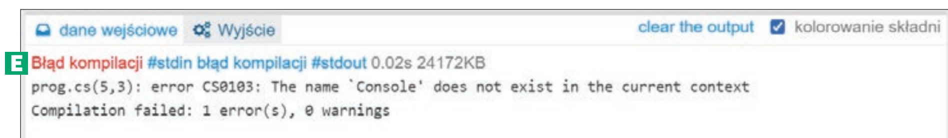
7 Spowoduje to ponowne uruchomienie napisanego przez nas programu, już z wprowadzonymi zmianami, które skutkują wystąpieniem błędów. W polu pod skryptem widać teraz wyjście danych, jednak brakuje w nim już oczekiwanego przez nas słowa **Testowanie**. Zamiast niego widzimy informację o wystąpieniu błędu kompilacji **E**. Błąd taki pojawia się, kiedy kompilator (czyli narzędzie, które kod źródłowy przetwarza na działający program) wykryje błąd w kodzie. W tym konkretnym przypadku błąd odnosi się do tego, że kompilator nie rozpoznaje, czym jest polecenie **Console.WriteLine("Testowanie");** – ponieważ nazwa **Console** nie znajduje się w przestrzeni nazw obsługiwanych przez ten program.

8 Nazwa **Console** znajduje się w przestrzeni nazw **System**, której to użycie usunęliśmy z naszego kodu źródłowego w poprzednich krokach. Czy istnieje zatem sposób na uniknięcie wystąpienia tego błędu, ale bez ponownego umieszczania w skrypcie **using System;**? Tak – przed poleceniem, które generuje wystąpienie błędu, należałoby użyć właśnie słowa **system** jako przestrzeni nazw, w której to znajduje się **Console**. Zatem linia kodu **Console.WriteLine("Testowanie");** powinna mieć postać **System.Console.WriteLine("Testowanie");** **F**.



9 Wprowadzoną zmianę skryptu możemy zatwierdzić, klikając na przycisk **wyślij** pod polem na skrypt. Zauważymy, że teraz na wyjściu danych z programu pojawi się już oczekiwane przez nas słowo **Testowanie** **G**.





10 Jeśli obydwa zapisy – zarówno z dyrektywą **using**, jak i bez niej – działają, a kod jej pozbawiony jest krótszy, dlaczego powinniśmy jej używać? Kod jest krótszy dlatego, że w obrębie naszych własnych poleceń użyliśmy jedynie polecenia wypisującego dane, a ponadto zrobiliśmy to tylko jednokrotnie. Gdyby w skrypcie pojawiło się więcej poleceń wymagających określenia przestrzeni nazw **System** – to za każdym razem od tego słowa powinniśmy rozpoczynać użycie takiego polecenia. A to znacznie wydłużyłoby nam kod źródłowy.

11 Jeżeli ponownie przyjrzymy się skryptowi z Unity, zauważymy, że dodanych do niego jest kilka przestrzeni nazw – każde użycie słowa **using** to dodanie kolejnej przestrzeni nazw. Wśród nich jest też using **UnityEngine;** **H**. To oznacza

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
```
H

dodanie do skryptu specyficznej dla silnika Unity przestrzeni nazw. W niej znajdują się polecenia, z jakich korzystamy tylko w połączeniu z silnikiem Unity. Próbując użyć tej przestrzeni nazw **I** na stronie **ideone.com**

</> kod źródłowy

```
using UnityEngine;

public class Test
{
    public static void Main()
```
I

i uruchamiając taki kod źródłowy, zostaniemy poinformowani o wystąpieniu błędu **J**, ponieważ nazwa tej przestrzeni nazw nie jest rozpoznawana przez kompilator.

informacja o kompilacji **J**

```
prog.cs(1,7): error CS0246: The type or namespace name `UnityEngine' could not be found. Are you missing an assembly reference?
Compilation failed: 1 error(s), 0 warnings
```

Zmienne w języku C#

Zmienne są ważnym elementem języków programowania. W dużym uproszczeniu można powiedzieć, że zmienne pozwalają na przechowanie w programie pod konkretną nazwą wartości potrzebnych do jego działania. Wartości te możemy odczytywać i – jak wskazuje na to nazwa – zmieniać.

Przykładem może być HP w grach, w których bohater może doznawać obrażeń. HP to punkty życia (od ang. health points). Ich liczba informuje, jak dużo zdrowia pozostało postaci. By taki mechanizm zaprogramować, zazwyczaj trzeba utworzyć zmienną o nazwie HP i przechowywać w niej aktualną liczbę punktów życia.

Wszystkie zmienne występujące w kodzie źródłowym programu w języku C# muszą zostać zadeklarowane jeszcze przed ich pierwszym użyciem.

Deklaracja zmiennej to określenie jej nazwy i typu. W języku C# wykorzystujemy różne typy zmiennych: liczbowe, tekstowe, znakowe oraz logiczne. Wśród typów liczbowych możemy wyróżnić liczby całkowite oraz zmiennoprzecinkowe, czyli takie, które pozwalają zapisać również wartości z częścią ułamkową. Każdy z typów zmiennych ma swoją indywidualną nazwę. Nazwy typów wraz z ich przeznaczeniem przedstawia tabelka obok.

Typy danych mogą być **wartościowe** – kiedy deklarujemy wartościowy typ danych, system od razu przydziela pamięć potrzebną na przechowywanie tej zmiennej. W innej gru-

WARTOŚCIOWE TYPY DANYCH

NAZWA TYPU	RODZAJ WARTOŚCI	OPIS	PRZYKŁAD UŻYCIA
int	Liczby całkowite	W zmiennych typu **int**, czyli **Integer**, można przechowywać liczby całkowite. W pamięci komputera na wartość takiej zmiennej rezerwowane są 32 bity. Ma to wpływ na zakres wartości, jakie może przyjmować zmienna tego typu. Są to liczby od -2 147 483 648 do 2 147 483 647. C# oferuje nam jednak jeszcze inne typy danych pozwalające na przechowywanie liczb całkowitych. Jeśli chcemy przechować liczbę spoza tego zakresu, możemy skorzystać z typu **long** – rezerwuje on 64 bity na zapisanie wartości i może się ona mieścić w zakresie od -9 223 372 036 854 775 808 do 9 223 372 036 854 775 807. Również dla mniejszych liczb są stosowane oddzielne typy – to **sbyte** (8 bitów) i **short** (16 bitów).	`public static void Main() { int x = 13456; }`
float	Liczby zmiennoprzecinkowe	**Float** to typ do przechowywania liczb zmiennoprzecinkowych. W takich zmiennych można przechowywać ułamki dziesiętne i ich przybliżenia, a także liczby z częścią ułamkową. Część ułamkową przechowywanej w zmiennej wartości zapisujemy po znaku kropki, samą wartość zaś kończymy, zapisując na końcu literę **f**. Na wartość zarezerwowane są 32 bity. Jeśli chcemy przechować liczbę zmiennoprzecinkową na większej liczbie bitów – 64, możemy użyć typu **double**. Typ **double** pozwala przechowywać więcej cyfr, dzięki czemu sprawdza się w wypadku liczb, które mają rozbudowaną część ułamkową, i pozwala na większą precyzję.	`public static void Main() { float x = 234.3214f; }`
char	Znak z tablicy Unicode	Taka zmienna przechowuje tylko jeden znak. Zapisujemy go w apostrofach.	`public static void Main() { char x = '2'; }`
bool	Typ logiczny	To specyficzny rodzaj zmiennej, która może przyjmować tylko jedną z dwóch wartości – **true** albo **false**.	`public static void Main() { bool x = true; }`

pie znajdują się typy **referencyjne**. Jednym z nich jest **typ łańcuchowy**, który pozwala nam na przechowywanie w zmiennej dowol-

```
public static void Main()
{
    string x = "napis";
}
```
A

nej wartości tekstowej. Wartość takiej zmiennej nadajemy, zapisując ją w cudzysłowie A.

Deklarując zmienne, powinniśmy przemyśleć dokładnie, jaki typ danych im nadawać. Przeanalizowanie kodu źródłowego B (patrz kolejna strona) pomoże nam zrozumieć logikę działania na zmiennych liczbowych.

1 W zaprezentowanym skrypcie mamy zmienną **x**, której wartość to 13. Jest to liczba całkowita, więc wykorzystano do jej przechowania typ **int**. Wynik dzielenia tej

B

```
using System;

public class Test
{
    public static void Main()
    {
        int x = 13;
        float y = x/2;
      C Console.WriteLine(y);
    }
}
```

wartości przez 2 powinien wynosić 6.5 – czyli jest to liczba z częścią ułamkową. Logiczne wydaje się zatem, że powinno wystarczyć zapisanie tego wyniku do zmiennej typu zmiennoprzecinkowego. Zgodnie z zaprezentowanym podejściem polecenie **Console.WriteLine(y);** C powinno skutkować wypisaniem wartości zmiennej, czyli 6.5.

2 Tak jednak nie jest. Co zatem pojawia się na wyjściu danych? Jest to liczba **6**. Dlaczego tak się dzieje? Dlatego że w działaniu dzielenia biorą udział dwie liczby całkowite, zapisane jako wartości całkowite.

```
dane wejściowe   Wyjście
Sukces #stdin #stdout 0.02s 22376KB
6
```

3 By w wyniku dzielenia pojawił się ułamek, choć jedna z nich powinna przyjąć typ **float**. Możemy to zrobić na dwa sposoby. Zapis **(float)x** D rzutuje wartość x na typ float, a zapis **2f** E – nadaje liczbie 2 typ float.

```
public static void Main()
{
    int x = 13;          D
    float y = (float)x/2;
    Console.WriteLine(y);
}
```

```
public static void Main()
{
    int x = 13;
    float y = x/2f;   E
    Console.WriteLine(y);
}
```

4 Użycie któregokolwiek z tych dwóch zapisów da nam już poprawny i oczekiwany wynik, czyli 6.5.

```
dane wejściowe   Wyjście
Sukces #stdin #stdout 0.02s 24604KB
6.5
```

Instrukcja warunkowa

Ważnym elementem programowania jest też instrukcja warunkowa. Pozwala ona na wykonywanie instrukcji w zależności od innych czynników, czyli od określonego warunku. Instrukcję warunkową tworzymy, pisząc **if**, a następnie w nawiasie zapisując warunek, który ma być spełniony. Jeśli w wypadku spełnienia warunku ma się wykonać tylko jedna instrukcja, możemy napisać ją bezpośrednio po nawiasie z warunkiem. I tak, zmieniając nieco analizowany wcześniej kod źródłowy, jeśli wartość zmiennej **y** miałaby pojawić się na wyjściu danych z programu tylko wtedy, kiedy jest ona większa niż 6, powinniśmy dopisać **if (y>6)** A przed **Console.WriteLine(y);**. Jeżeli pozostawimy kod bez zmian, na wyjściu nadal będzie pojawiać się wartość 6.5, ponieważ jest ona większa niż 6.

```
public static void Main()
{
    int x = 13;
    float y = x/2f;
    if (y>6) Console.WriteLine(y);
}       A
```

Jednak gdybyśmy podzielili x nie przez 2f, ale przez 3f, wynik, jaki trafiłby do zmiennej **y**, to 4. Wartość ta jest mniejsza niż 6, zatem warunek nie byłby spełniony – na wyjściu danych z programu nie pojawi się nic.

Wykonanie bloku instrukcji

Jeśli w wypadku spełnienia warunku chcemy, by wykonało się więcej instrukcji, blok instrukcji do wykonania umieszczamy w nawiasie klamrowym – **{}**. Oto przykład B wykonania dwóch instrukcji wypisania danych w przypadku spełnienia warunku.

ŁĄCZENIE WARUNKÓW

Warunki do sprawdzenia możemy łączyć. Jeśli chcemy, by jakieś instrukcje wykonały się tylko wtedy, gdy spełnione są dwa warunki, możemy te dwa warunki zapisać w jednej instrukcji warunkowej, w tym samym nawiasie – umieszczając pomiędzy nimi **&&** **A**.

Jeżeli chcemy, by jakieś instrukcje wykonały się, gdy spełniony będzie choć jeden

```
if (y>6 || x>5)
```

z dwóch warunków, możemy połączyć je znakami **||**.

```
if (y>6 && x>5)
{        A
    Console.WriteLine("Wartość y jest większa niż 6, a x większa od 5.");
}
```

Więcej warunków

W jednej instrukcji warunkowej może znaleźć się więcej niż jeden warunek. Różne instrukcje mogą być wykonane, gdy inne warunki są spełnione. Kolejne warunki umieszcza się w nawiasach po **else if** **C**. Są one sprawdzane, gdy nie są spełnione wcześniejsze warunki.

B
```
int x = 13;
float y = x/2f;
if (y>6)
{
    Console.WriteLine(y);
    Console.WriteLine("Wartość y była większa niż 6");
}
```

```
if (y>6)
{
    Console.WriteLine(y);
    Console.WriteLine("Wartość y była większa niż 6");
}
else if(y > 4) C
{
    Console.WriteLine(y);
    Console.WriteLine("Wartość y była większa niż 4");
}
```

W przeciwnym razie

Instrukcja warunkowa może (choć nie musi) mieć też sekcję **else** **D**. W niej opisujemy, co ma się wykonać, gdy żaden z wcześniej opisanych w instrukcji warunków nie został spełniony.

```
if (y>6)
{
    Console.WriteLine(y);
    Console.WriteLine("Wartość y była większa niż 6");
}
else if(y > 4)
{
    Console.WriteLine(y);
    Console.WriteLine("Wartość y była większa niż 4");
}
else D
{
        Console.WriteLine("Wcześniej sprawdzane warunki nie zostały spełnione");
}
```

Instrukcja Switch

Switch to instrukcja wyboru. Pozwala ona na sterowanie programem w zależności od wartości wybranej zmiennej. Instrukcji tej blisko jest do instrukcji warunkowej – **if**, jednak tu nie sprawdzamy całego warunku, ale bierzemy pod uwagę jedynie możliwe

wartości operatora, czyli zmiennej sterującej tą instrukcją.

1 Nazwę zmiennej zapisujemy w nawiasie po słowie **switch**.

```
switch (y)
```

2 Dalej w nawiasie klamrowym umieszczamy kolejne sekcje **case**, po których piszemy możliwą wartość zmiennej sterującej i znak dwukropka.

```
switch (y)
{
    case 1:

    case 2:

    case 3:
}
```

3 W kolejnych liniach po linii z **case** znajdują się instrukcje, jakie mają się wykonać, gdy zmienna sterująca ma podaną wartość.

```
switch (y)
{
    case 1:
        Console.WriteLine("Measured value is");
        Console.WriteLine("Measured value is");

    case 2:
        Console.WriteLine("Measured value is");
        Console.WriteLine("Measured value is");
```

```
case 1:
    Console.WriteLine("Measured value is");
    Console.WriteLine("Measured value is");
    break; A
case 2:
    Console.WriteLine("Measured value is");
    Console.WriteLine("Measured value is");
    break;
case 3:
    Console.WriteLine("Measured value is");
    Console.WriteLine("Measured value is");
    break;
```

4 Blok tych instrukcji kończy słowo kluczowe **break;** A – po którym może znaleźć się kolejny case itd.

5 A co w sytuacji, gdy zmienna ta ma inną wartość? Wtedy dopisujemy oprócz kolejnych sekcji case sekcję **default** B – zawarte w niej instrukcje wykonają się przy innej wartości zmiennej sterującej.

Ponieważ sprawdzana jest pojedyncza zmienna w zdefiniowanych przypadkach (case), to musi ona mieć jednoznaczną wartość liczby całkowitej lub wartość dającą się przedstawić za pomocą liczby całkowitej. To wszystko sprawdza się do tego, że zmienna użyta w instrukcji jako zmienna sterująca musi mieć odpowiedni typ, czyli na przykład **int** C, **short** czy **long**. Niedozwolone jest za to stosowanie typów **float** i **double**, ponieważ ze względu na przybliżenia sprawdzanie warunków mogłoby nie być jednoznaczne, a co za tym idzie, zastosowanie takiej zmiennej jako sterującej będzie skutkowało występowaniem błędu D podczas kompilacji kodu.

```
    break;
default: B
    Console.WriteLine("Nie wykryto sprawdzanej wartości");
    break;
```

```
int y = 3; C
switch (y)
{
    case 1:
        Console.WriteLine
```

```
dane wejściowe   Wyjście                clear the output   kolorowanie składni
Błąd kompilacji #stdin błąd kompilacji #stdout 0.02s 26116KB D
prog.cs(8,3): error CS0151: A switch expression of type `float' cannot be converted to an integral type,
bool, char, string, enum or nullable type
Compilation failed: 1 error(s), 0 warnings
```

Pętle while i do while

Mamy więcej mechanizmów warunkowych w programowaniu, które uzależniają wykonywanie pewnych instrukcji od warunków, czy też wartości konkretnych zmiennych. Do tego typu mechanizmów należą też pętle warunkowe, które nie tylko uzależniają wykonanie instrukcji od spełnienia warunku, ale też potrafią te instrukcje wykonywać wielokrotnie.

Można powiedzieć, że pętle warunkowe stosuje się wtedy, kiedy nie wiemy, jak wiele razy trzeba będzie w programie powtórzyć zapętloną czynność. Pętle warunkowe w języku C# mamy dwie – **while** i **do while**. W tej części rozdziału poznamy obydwie.

While

Pętli **while** blisko jest do instrukcji warunkowej – nawet w kwestii składni. Oto przykładowy kod źródłowy zawierający instrukcję warunkową **if** A.

</> kod źródłowy

```
using System;

public class Test
{
    public static void Main()
    {
        int x = 8;
   A    if (x <= 10)
        {
            Console.WriteLine(x); B
            x = x + 1; C
        }
    }
}
```

1 Mamy tu zmienną o nazwie **x**, której nadaliśmy wartość 8.

2 Poprzez instrukcję warunkową sprawdzamy, czy wartość zmiennej **x** jest mniejsza lub równa 10. Jeśli tak – wypisujemy wartość zmiennej **x** B na wyjściu danych i następnie zwiększamy ją o 1 C.

3 Instrukcja warunkowa wykonuje się tylko raz. Raz sprawdzany jest warunek i dla przedstawionych danych jest on spełniony, więc instrukcje z wnętrza nawiasu klamrowego wykonują się również raz. Zatem na wyjściu danych z programu pojawia się wartość zmiennej **x**, czyli 8.

```
dane wejściowe   Wyjście
Sukces #stdin #stdout 0.02s 24444KB
8
```

4 Gdybyśmy zastąpili słowo **if** słowem **while** D, działanie programu nieco by

```
   int x = 8;
D  while (x <= 10)
   {
       Console.WriteLine(x);
       x = x + 1;
   }
```

się zmieniło. To dlatego, że po zakończeniu wykonywania bloku instrukcji zamkniętego w klamrowy nawias program powróciłby do sprawdzania warunku z nawiasu. Jeśli nadal byłby on spełniony, ponownie wykonałyby się instrukcje z nawiasu klamrowego. Czynność ta byłaby powtarzana aż do momentu, gdy warunek nie będzie spełniony. W przypadku zaprezentowanej pętli na wyjściu pojawiłyby się liczby 8, 9 i 10, a pętla wykonałaby się trzy razy.

```
dane wejściowe   Wyjście
Sukces #stdin #stdout 0.02s 24356KB
8
9
10
```

Do while

Innym rodzajem pętli warunkowej jest pętla **do while**, która od wcześniej omawianej różni się przede wszystkim momentem sprawdzania warunku. W tej pętli najpierw wykonywane są zapętlone instrukcje, a po ich wykonaniu sprawdzany jest warunek. Jeśli jest on spełniony, instrukcje z pętli wykonywane są ponownie. To oznacza, że nawet jeśli przystępujemy do pętli z warunkiem, który nie jest spełniony jeszcze przed jej startem, instrukcje z nawiasu klamrowego i tak wykonają się przynajmniej jeden raz.

1 Tworząc pętlę, rozpoczynamy ją od słowa **do**, po którym umieszczamy nawias klamrowy.

```
do
{

}
```

2 W nawiasie klamrowym powinien znaleźć się blok instrukcji E do wykonania w pętli.

```
do
{
E   Console.WriteLine(x);
    x = x + 1;
}
```

3 Po nawiasie klamrowym pojawia się dopiero słowo **while** F wraz z nawiasem zawierającym warunek wykonywania pętli.

```
do
{
    Console.WriteLine(x
    x = x + 1;
}   while (x <= 10); F
```

4 Dla przedstawionej tu pętli warunek mówi o tym, że x musi być mniejsze od 10 lub równe 10. Jeśli zmienna **x** miałaby wartość, która nie spełnia tego warunku – na przykład 80 G – to pętla i tak wykonałaby się, ponieważ najpierw wykonywane

```
int x = 80; G
do
{
```

są instrukcje, a potem dopiero sprawdzane jest spełnienie warunku. Po jednokrotnym wykonaniu instrukcji warunek nie jest spełniony – pętla przerywa swoje działanie. Na wyjściu danych z tak skonstruowanego programu pojawi się tylko jedna liczba – 80 – jako wartość zmiennej **x**.

```
dane wejściowe   Wyjście
Sukces #stdin #stdout 0.02s 22316KB
80
```

Pętla for

Pętla **for**, nazywana też pętlą ogólną, to konstrukcja programistyczna stanowiąca jeden z rodzajów pętli, który jest dostępny w wielu językach programowania. Nawet jeśli chodzi o składnię, pętla ta będzie wyglądała podobnie zarówno w języku C#, jak i C++, Java czy nawet JavaScript. Pętla ta umożliwia między innymi definiowanie pętli iteracyjnych, czyli takich, które wykonują się określoną liczbę razy.
Pętlę tworzymy, używając słowa **for** A, a następnie w nawiasie podając trzy elementy budujące pętlę i oddzielając je średnikami.

```
public static void Main()
{
  A for ()
    {

    }
}
```

1 Pierwszy z nich to element inicjalizacji. Wykonywany jest w pierwszej kolejności i w przeciwieństwie do pozostałych dwóch elementów budujących pętlę – tylko raz. Krok ten pozwala na zdeklarowanie zmiennej, która steruje działaniem pętli. Dlatego jest ona nazywana zmienną sterującą. Na potrzeby tej pętli najczęściej deklaruje się zmienną o nazwie **i** w typie **int** z wartością początkową równą **0**.

```
for (int i = 0;)
{

}
```

2 Drugi element to warunek. Jeżeli jest prawdziwy, blok kodu wewnątrz pętli zostanie wykonany. Jeśli nie – pętla się zakończy, a program automatycznie przejdzie do dalszej części kodu znajdującej się bezpośrednio po pętli for. Zazwyczaj do budowy tego warunku wykorzystuje się zmienną stworzoną na potrzeby pętli w pierwszej części, czyli w elemencie inicjalizacji. Taki warunek to może być na przykład **i < 10**.

```
for (int i = 0; i < 10;)
{

}
```

3 Trzeci i ostatni z elementów budujących pętlę wykonywany jest dopiero po wykonaniu całej zawartości pętli. Element ten jest też nazywany elementem inkrementacji. W nim możemy dokonać aktualizacji wartości zmiennej sterującej. Najczęściej z każdym przejściem pętli wartość tej zmiennej zwiększa się o 1. Zwiększenie o 1 zmien-

DEKREMENTACJA

Zapis dwóch plusów po zmiennej nazywamy inkrementacją – to operacja zwiększania wartości o 1.
A jeśli chodzi o zmniejszanie wartości zmiennej, mamy analogiczną operację, którą nazywamy dekrementacją. Znaki plus zastępujemy w niej znakami minus, zatem zapis **i--** zmniejsza o 1 wartość zmiennej **i**. Dla zaprezentowanego przykładu zmienna **i** po wykonaniu polecenia **i--** będzie miała już wartość **5**.

```
int i = 6;
i--;
```

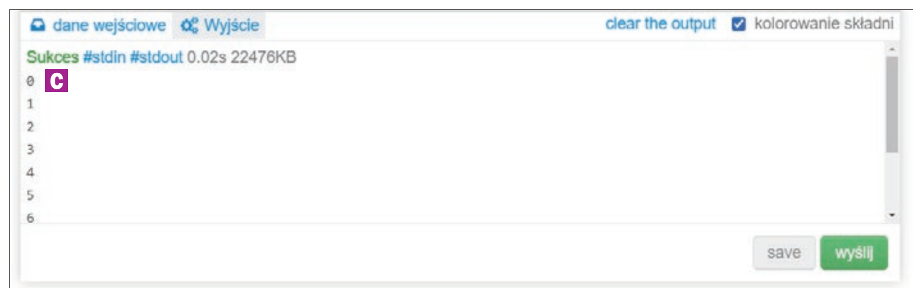
nej **i** moglibyśmy zapisać jako **i = i + 1**. C# pozwala jednak na nieco skrócony zapis tej operacji – ponieważ ten sam efekt uzyskamy, zapisując w naszym kodzie **i++**.

```
for (int i = 0; i < 10; i++)
```

4 Podobnie jak w przypadku wcześniej opisanych pętli, w pętli **for** instrukcje, które mają się powtarzać, także zapisujemy w nawiasie klamrowym. Po kolejnym wykonaniu ponownie sprawdzany jest warunek – i tak aż do momentu, kiedy przestanie on być spełniany, są powtarzane instrukcje z wnętrza pętli. Co ważne, ze zmiennej sterującej możemy korzystać we wnętrzu pętli – w zaprezentowanym przykładzie umieszczono w pętli instrukcję wypisującą **B** wartość zmiennej sterującej.

```
for (int i = 0; i < 10; i++)
{
    Console.WriteLine(i); B
}
```

5 Na wyjściu danych z programu z każdym przejściem pętli pojawia się większa liczba – począwszy od 0 **C**. Przy tak skonstruowanej pętli ostatnia wypisana wartość, jaka pojawi się na wyjściu danych, to 9. Jeśli zmienna sterująca przy deklaracji dostaje wartość 0, a z każdym przejściem pętli wartość tej zmiennej rośnie o 1, to jeśli w warunku sprawdzamy, czy zmienna ta ma wartość mniejszą niż pewna określona liczba – to liczba ta jednocześnie mówi nam o tym, ile razy wykona się pętla. Dla zaprezentowanego przykładu było to 10 powtórzeń.



Pętla foreach i tablice

Kolejnym z rodzajów pętli, jakie możemy wykorzystać podczas programowania w języku C#, jest pętla **foreach**, która pozwala na wykonanie bloku instrukcji dla każdego elementu zbioru. By móc dobrze zrozumieć działanie tej pętli, musimy wiedzieć, że zmienne nie są jedyną strukturą w języku C#, jaką można wykorzystać do

przechowywania danych. Czasami zmienne mogą nie wystarczyć do przechowania danych przetwarzanych przez program. W jednej zmiennej może znajdować się w danej chwili tylko jedna wartość. W przypadku gdy program powinien przechowywać wiele wartości o podobnym znaczeniu, trzeba byłoby zadeklarować wiele zmiennych. Przykładem mogą być długości skoków uzyskane przez skoczków narciarskich w zawodach. W takiej sytuacji konieczne byłoby stworzenie osobnej zmiennej dla każdego skoczka i przypisanie każdej z nich innej wartości odpowiadającej długości skoku uzyskanej przez danego zawodnika.

Tablice

Do przechowywania określonej liczby elementów tego samego typu lepiej wykorzystać strukturę danych zwaną **tablicą**. Podobnie jak w wypadku zmiennej, trzeba ją najpierw zadeklarować, czyli określić jej nazwę i typ przechowywanych danych, a dodatkowo trzeba także podać jej rozmiar, czyli to, ile elementów ma się w niej znaleźć. W przypadku tablicy przechowującej długości skoków uzyskane przez skoczków moglibyśmy zastosować typ danych **float** i stworzyć tablicę o nazwie **odleglosci** zawierającą **50** elementów.

1 Tworząc taką tablicę, po nazwie typu umieszczamy pusty nawias kwadratowy **A**, dalej podajemy nazwę, a po znaku równości używamy słowa **new** i ponownie podajemy nazwę typu danych zakończoną nawiasem kwadratowym, tym razem jednak nie pustym, ale wypełnionym liczbą elementów tworzonej tablicy.

```
public static void Main()
{
    float[] odleglosci = new float[50];
}
```
A

2 Odnosząc się do poszczególnych elementów tablicy, używamy ich numerów, nazywanych też indeksami. Elementy te numerowane są od zera, zatem pierwszy element zadeklarowanej w przykładzie tablicy miałby numer 0, a ostatni 49. Aby odwołać się do jakiegoś elementu tablicy, należy po nazwie tablicy w nawiasie kwadratowym umieścić indeks, czyli numer elementu.

3 Zatem zapis **odleglosci[23]** pozwoli nam na odniesienie się do elementu tablicy o indeksie 23.

Pętla foreach

Dysponując zbiorem elementów – na przykład opisaną wcześniej tablicą – możemy skonstru-

TYP ŁAŃCUCHOWY JAKO TABLICA

Deklarując zmienną typu łańcuchowego – **string**, tak naprawdę tworzymy tablicę znaków, czyli tablicę elementów typu **char**. Jest to o tyle przydatne rozwiązanie, że do poszczególnych znaków w zmiennej typu **string** możemy się odnosić jak do elementów tablicy, czyli poprzez ich indeks.

1 Zakładając, że mamy zmienną o nazwie **napis** z wartością „KomputerSwiat” **A**, możemy użyć zapisu **napis[1]** **B**, który będzie wskazywał na drugi znak w ciągu znaków (elementy tablicy są numerowane od zera).

```
public static void Main()
{
    string napis = "KomputerSwiat";
    Console.WriteLine(napis[1]);
}
```
A, B

2 W zaprezentowanym przykładzie byłaby to litera **o** i to właśnie ją **C** widzimy na wyjściu danych z programu, gdy chcemy uzyskać wartość **napis[1]**.

dane wejściowe | Wyjście

Sukces #stdin #stdout 0.02s 22164KB

o **C**

ować pętlę, która wykona się tyle razy, ile elementów jest w zbiorze, a zmienna sterująca będzie jako wartość przyjmowała kolejne elementy zbioru. Tak działa pętla **foreach**. Dysponując pięcioelementową tablicą liczb całkowitych (wartości elementom tablicy możemy nadać bezpośrednio po jej deklaracji, umieszczając kolejne elementy tablicy w nawiasie klamrowym B), możemy skonstruować pętlę **foreach**, która wykona się pięć razy.

```
public static void Main()
{
  B int[] wyniki = new int[5] {4,6,2,2,9};
}
```

1 Tworząc pętlę, używamy słowa **foreach**, po którym umieszczamy nawias, w którym dalej znajdzie się zakres wykonywania pętli.

```
int[] wyniki = new int[5] {4,6,2,2,9};
foreach ()
```

2 W zakresie wykonywania pętli deklarujemy zmienną sterującą – na przykład **int x** C, a dalej używamy słowa **in** i podajemy nazwę zbioru, dla którego zmienna sterująca w kolejnych przejściach pętli ma przyjmować kolejne wartości.

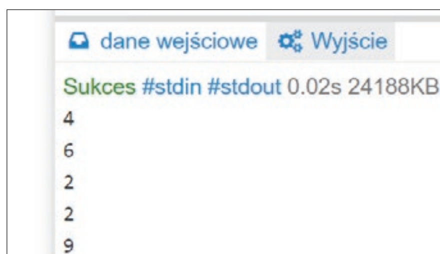
3 Podobnie, jak miało to miejsce w pozostałych pętlach, jakie omawialiśmy już w tym rozdziale, blok instrukcji, które mają wykonać się wiele razy, umieszczamy w nawiasie klamrowym **{}** D.

```
foreach (int x in wyniki)
{            C
    | D
}
```

4 Jeśli w pętli umieścimy instrukcję wypisującą wartość zmiennej **x** E na wyjściu

```
foreach (int x in wyniki)
{
    Console.WriteLine(x); E
}
```

danych z programu, to powinny pojawić się tam kolejne wartości znajdujące się w tablicy.



Funkcje

Kolejnym ważnym elementem języków programowania, w tym języka C#, są funkcje. Temat funkcji rozpoczęliśmy omawiać już przy okazji tworzenia menu gry w poprzednim rozdziale tej książki. Po pierwsze – skasowaliśmy ze skryptu dwie domyślnie obecne w nim funkcje **Start** i **Update**, a po drugie – napisaliśmy własną funkcję do zamykania gry. Na tej podstawie moglibyśmy już intuicyjnie określić, czym są funkcje. Jednak to, czego do tej pory dowiedzieliśmy się na temat funkcji, to tylko mała część całego zagadnienia.

Jeśli w wielu miejscach kodu źródłowego trzeba użyć tego samego fragmentu kodu, można utworzyć z niego funkcję, którą inaczej nazywamy podprogramem. Następnie w każdym miejscu, w którym chcemy skorzystać z tego fragmentu, wystarczy go wywołać, podając jego nazwę, dzięki czemu nie trzeba go znowu pisać. Co za tym idzie, kod źródłowy jest krótszy, czytelniejszy, a jego tworzenie szybsze.

Tworzenie funkcji

Funkcje budujemy zawsze w ten sam sposób:

1 Zaczynamy od określenia typu danych, które będzie zwracać funkcja. Szczególnym rodzajem funkcji jest funkcja **void**, która nie zwraca żadnej wartości. Tego typu funkcje były do tej pory prezentowane i tego typu funkcje są przydatne w Unity. Wygodnie jest przypisywać je do zdarzeń – stanowią wtedy zwykły wydzielony fragment kodu.
Teraz stwórzmy jednak inną funkcję – niech będzie to funkcja zwracająca dane w typie **bool** A. Przed nazwą typu użyjmy słowa **static**.

```
static bool czyKwadratParzysty(int x)
          A                        B
public static void Main()
{

}
```

2 Każda funkcja musi mieć swoją unikalną nazwę. Dzięki temu będzie ją można wywołać w dowolnym miejscu kodu, ale też – w przypadku Unity – odnieść się do niej z poziomu edytora.

3 Funkcja może też przyjmować parametry. Parametry określają, jakie dane i jakiego typu przyjmuje funkcja. Parametrów może być kilka – wymieniamy je, oddzielając je przecinkami. Zakładając, że chcemy stworzyć funkcję sprawdzającą, czy kwadrat podanej w parametrze liczby jest liczbą parzystą, w parametrze funkcji powinna się znaleźć liczba całkowita – w naszym przykładzie liczbę tę nazwiemy x B.

4 Instrukcje tworzące treść funkcji umieszczamy wewnątrz nawiasów klamrowych C, które są nieodłącznym elementem języka C#.

```
static bool czyKwadratParzysty(int x)
{
    | C
}
```

5 W przypadku opisywanej funkcji, by sprawdzić parzystość kwadratu liczby x, najpierw możemy ten kwadrat policzyć. I tak, wewnątrz funkcji możemy zadeklarować zmienną **y**, w której zapiszemy ten kwadrat, czyli wynik mnożenia x przez x D. Co warte podkreślenia, gdy deklarujemy zmienną w obrębie funkcji, jest ona dostępna tylko w funkcji. Gdybyśmy poza treścią funkcji chcieli wypisać wartość zmiennej **y**, byłoby to niemożliwe.

```
static bool czyKwadratParzysty(int x)
{
    int y = x*x;| D
}
```

6 Dysponujemy już liczbą, której parzystość chcemy sprawdzić, powinniśmy teraz zastanowić się, jak tego dokonać. Liczba jest parzysta, jeśli jej reszta z dzielenia przez 2 jest równa 0. W języku C# dysponujemy narzędziem do obliczania reszty z dzielenia. Za resztę z dzielenia odpowiada znak %. Zapis **y%2** E zwraca resztę z dzielenia wartości zmiennej **y** przez 2. Jeśli ten wynik jest równy 0 – y jest liczbą parzystą.

MIEJSCE DEKLARACJI FUNKCJI

Zwróćmy uwagę na to, że funkcja została zadeklarowana w nieco innym miejscu kodu, niż wcześniej pisane przez nas instrukcje. Wcześniejsze fragmenty wpisywaliśmy nieco niżej – we wnętrzu **funkcji Main**. To dlatego, że w podstawowym schemacie skryptu znajduje się już jedna funkcja – właśnie o nazwie **Main**. Pełni ona rolę tak zwanej **funkcji głównej programu**. Jest to funkcja, która jest wywoływana wraz ze startem programu. Jako że nie deklarujemy funkcji wewnątrz innej funkcji, należałoby to zrobić przed nią. W przypadku skryptów tworzonych w Unity nie mamy tego problemu – skrypty nie są uruchamiane wraz ze startem programu, są tylko jego elementami – zatem nie ma w nich funkcji głównej.

```
static bool czyKwadratParzysty(int x)
{
    int y = x*x;
    if (y%2 == 1) E
    {
        return false;
    }    F     G
    else
    {
        return true;
    }          G
}
```

7 Wynik działania funkcji zostanie zwrócony dzięki instrukcji **return** F. Po niej podaje się zwracaną przez funkcję wartość. Moment i rodzaj zwrócenia wartości może być określony przez instrukcję warunkową – zależnie od wyniku obliczenia reszty z dzielenia.

8 W przypadku funkcji, której typem jest **bool**, może ona zwracać jedynie wartości **true** i **false** G.

Wywoływanie funkcji

By sprawdzić efekt działania funkcji, należałoby ją wywołać, używając przy tym parametru, który będzie miał wpływ na wynik działania funkcji. Wywołujemy funkcję, umieszczając w kodzie jej nazwę, a po niej w nawiasie podając wartość parametru lub nazwę zmiennej, w której ta wartość jest zapisana. Jednak samo wywołanie funkcji niesie za sobą pewną wartość. By móc zobaczyć, jaka jest to wartość, należy ją wypisać na wyjściu danych.

1 Zatem wywołanie funkcji H można umieścić w poleceniu wypisującym dane na wyjściu danych z programu. Zwróćmy też uwagę na miejsce wywołania funkcji – znajduje się już w funkcji głównej.

```
public static void Main() H
{
    Console.WriteLine(czyKwadratParzysty(9));
}
```

OPERATOR RÓWNOŚCI

O ile znaki mniejszości i większości użyte w warunkach są dość intuicyjne, zastanawiającą może być dla nas obecność dwóch znaków równości w instrukcji warunkowej. To nie jest błąd. W ten sposób zapisujemy operator równości. Jeden znak równości oznacza instrukcję przypisania wartości.
Jeśli z kolei chcemy w warunku określić, że jest on spełniony, gdy pewna wartość jest różna od innej, możemy użyć operatora nierówności, czyli wykrzyknika i znaku równości.

2 Kwadratem liczby 9, której użyto w przykładzie jako parametru funkcji, jest 81. Ta liczba jest liczbą nieparzystą, zatem oczekiwanym wynikiem działania funkcji będzie

```
dane wejściowe   Wyjście
Sukces #stdin #stdout 0.02s 22204KB
False
```

False i taka też wartość pojawia się na wyjściu danych z programu.

3 By upewnić się co do prawidłowości działania funkcji, uruchommy ją z parametrem takim, by oczekiwanym wynikiem działania funkcji było **True**. Takim parametrem może być wartość 8 I, ponieważ jej kwadrat to 64 – czyli liczba parzysta.

```
public static void Main()                      I
{
    Console.WriteLine(czyKwadratParzysty(8));
}
```

4 Dla takiego parametru nasza funkcja zwraca wartość **True**.

```
dane wejściowe   Wyjście
Sukces #stdin #stdout 0.02s 24128KB
True
```

Przećwicz to!

By lepiej poznać tajniki programowania w języku C#, wykonajmy kilka samodzielnych zadań. Ich rozwiązania znajdują się poniżej. Zajrzyjmy do nich dopiero po podjęciu próby samodzielnego rozwiązania.

Zadanie 1

Napisz program, który na wyjściu danych wypisze wartości podniesionych do sześcianu liczb ze zbioru: 83,23,43,28,47,90.

Zadanie 2

Napisz program zawierający funkcję, która zlicza wystąpienia określonego znaku w podanym ciągu znaków.

Zadanie 3

Napisz program, który dla podanego zestawu liczb wskazuje największą i najmniejszą z nich.

Rozwiązania

Oto propozycje rozwiązań zaprezentowanych wyżej zadań. Pamiętajmy jednak, że w programowaniu jest tak, że każde zadanie można rozwiązać na kilka sposobów. W tych rozwiązaniach pokazano tylko jeden z możliwych sposobów.

Rozwiązanie zadania 1

1 Jeśli program ma pracować na zbiorze danych, możemy ten zbiór danych zapisać w formie tablicy. W przedstawionym zbiorze mamy tylko liczby całkowite, więc typem danych będzie **int**. Zbiór zawiera sześć elementów – tyle też będzie liczyć tablica utworzona na potrzeby jego przechowania. Wartości elementów tablicy możemy zapisać, podając je w nawiasie klamrowym **A** po deklaracji tablicy.

```
public static void Main()
{                                        A
    int[] liczby = new int[6] {83,23,43,28,47,90};
}
```

2 Mamy sześcioelementową tablicę. By odnieść się do każdego z jej elementów, możemy skonstruować pętlę **for**, która pozwoli na sześciokrotne wykonanie umieszczonych w niej instrukcji. Zatem na potrzeby pętli deklarujemy zmienną **i** o początkowej wartości **0** **B**, która będzie zwiększała się z każdym przejściem pętli o 1, dopóki jej wartość będzie mniejsza niż 6.

```
int[] liczby = new int[6] {83,23,43,28,47,90};
for (int i=0; i < 6; i++)
{         B

}
```

3 Mając zbudowaną pętlę, w jej wnętrzu poprzez zapis **liczby[i]** z każdym przejściem pętli odnosimy się do kolejnej liczby ze zbioru. Taką liczbę należałoby podnieść do sześcianu – czyli pomnożyć przez samą siebie i potem ponownie pomnożyć przez samą siebie. Wynik takiego mnożenia powinien być wypisany na wyjściu danych z programu poprzez polecenie **Console.WriteLine** C.

```
for (int i=0; i < 6; i++)
{
    Console.WriteLine(liczby[i]*liczby[i]*liczby[i]);
}
```
C

4 Po uruchomieniu tak napisanego programu na wyjściu danych powinny pojawić się już liczby, które odpowiadają sześcianom liczb z podanego na początku programu zbioru.

```
dane wejściowe   Wyjście
Sukces #stdin #stdout 0.02s 26336KB
571787
12167
79507
21952
103823
729000
```

Rozwiązanie zadania 2

1 W drugim zadaniu należało posłużyć się definicją funkcji – tę umieszczamy poza funkcją główną programu. Jeśli funkcja ma zwracać liczbę znaków z ciągu znaków – to będzie to liczba całkowita, zatem typem zwracanych danych powinien być **int** A.

```
static int zliczanie()

public static void Main()
{

}
```
A

2 Funkcja powinna zwracać nam inny wynik w zależności od tego, jakie podamy w niej słowo (ciąg znaków) i jaki znak będziemy zliczać. Zatem te dwie rzeczy powinny trafiać do funkcji poprzez jej parametry. Jeden parametr to ciąg znaków – typ **string**, a drugi to pojedynczy znak (typ **char** B).

```
static int zliczanie(string slowo, char znak)
```
B

3 Jeśli funkcja ma coś zliczać, to można w jej obrębie wykorzystać zmienną o nazwie **licznik**, w której będziemy magazynować zliczone znaki. Zmienna ta powinna przechowywać liczbę całkowitą, a jej wartość początkowa to 0 C.

```
static int zliczanie(string s
{
    int licznik = 0;
}
```
C

4 Dalej nasza funkcja powinna brać kolejne znaki w ciągu znaków i porównywać je ze znakiem, dla którego mamy określić liczbę wystąpień. Możemy tu wykorzystać właściwość wartości typu **string**, to że są one jak tablice, i skonstruować pętlę **foreach** D, która wykona się dla każdego znaku w podanym ciągu. Każdy znak w obrębie pętli będzie rozpatrywany jako wartość **z** typu **char**.

```
int licznik = 0;
foreach (char z in slowo)
{

}
```
D

5 Wewnątrz pętli musimy ten znak porównywać ze zliczanym znakiem. Porównanie może odbywać się poprzez instrukcję warunkową – **if** E.

```
foreach (char z in slowo)
{
    if (z == znak)
    {

    }
}
```
E

6 Gdy porównanie zwraca nam informację o wykryciu poszukiwanego znaku, powinniśmy zwiększyć wartość **licznika** F magazynującego liczbę wystąpień znaku.

```
if (z == znak)
{
    licznik++;
}
```
F

7 Po przejściu całej pętli **foreach** wiemy, że porównanie zostało wykonane dla

wszystkich znaków w ciągu. Możemy zatem zwrócić wartość licznika poprzez użycie **return licznik;**.

```
                licznik++;
            }
        }
        return licznik;
}
```

8 By sprawdzić, jak działa nasza funkcja, powinniśmy ją wywołać – co powinno mieć miejsce już w funkcji głównej. Wynik jej działania będzie liczbą całkowitą, więc aby go zobaczyć, można wypisać (**Console.**

```
public static void Main()
{
    G Console.WriteLine(zliczanie("Kleopatra", 'a'));
}
```

WriteLine G) wartość tej liczby na wyjściu danych z programu.

9 Przy wywołaniu naszej funkcji dla słowa „Kleopatra" i zliczaniu wystąpień w nim znaku „a" uzyskamy oczekiwany rezultat, czyli liczbę **2**.

```
dane wejściowe   Wyjście
Sukces #stdin #stdout 0.02s 22384KB
2
```

Rozwiązanie zadania 3

1 Trzecie z zadań wymagało określenia najmniejszej i największej wartości w zbiorze liczb. Zbiór ten nie był określony, zatem na potrzeby przykładu został wybrany zbiór ośmiu liczb i taka też tablica ośmiu liczb całkowitych A została zadeklarowana.

```
public static void Main()                                   A
{
    int[] zestaw = new int[8] {54,32,64,26,74,765,43,90};
}
```

2 Jeśli mamy tablicę liczb, to możemy odnosić się do poszczególnych jej elementów poprzez pętlę **foreach** B i tak też zostało to zrobione w przykładowym rozwiązaniu tego zadania.

```
  int[] zestaw = new int[8] {54,32,64,26,74,765,43,90};
B foreach (int liczba in zestaw)
  {

  }
```

3 Zadanie polega na określeniu maksymalnej i minimalnej wartości w tym zbiorze. Obydwie te wartości zapiszemy do zmiennych. Deklarując je, powinniśmy nadać zmiennym takie wartości, aby maksimum C

```
int[] zestaw = new int[8] {54,
int minimum = 123456789; C
int maksimum = -123456789; D
foreach (int liczba in zestaw
```

miało wartość bardzo małą, a minimum D – bardzo dużą.

```
foreach (int liczba in zestaw)
{
  E if (liczba > maksimum) maksimum = liczba;
}
```

4 Wewnątrz pętli powinniśmy sprawdzać, czy każda kolejna liczba jest większa niż zapisane przez nas w zmiennej maksimum. Możemy to zrobić poprzez instrukcję warunkową – **if** E. Gdy taka sytuacja ma miejsce, wartość aktualnie sprawdzanej liczby powinna być zapisana jako nowa wartość naszego maksimum.

5 Również wewnątrz pętli analogiczny mechanizm powinniśmy zastosować dla najmniejszej liczby w zbiorze. Kolejne liczby powinny być porównywane z aktualnie zapisanym minimum F – jeśli liczba jest od niego mniejsza – jej wartość powinna zastąpić aktualnie przechowywaną wartość minimalną ze zbioru.

```
foreach (int liczba in zestaw)
{
    if (liczba > maksimum) maksimum = liczba;
  F if (liczba < minimum) minimum = liczba;
```

6 Po przejściu całej pętli w naszych zmiennych powinny być już zapisane wartości maksymalna i minimalna. Można zatem wypisać je na wyjściu danych z programu. W poleceniu **Console.WriteLine** możemy

```
        if (liczba < minimum) minimum = liczba;
    }                                       G
    Console.WriteLine("Największa z liczb to:" + maksimum);
    Console.WriteLine("Najmniejsza z liczb to:" + minimum);
}
```

w nawiasie posłużyć się dodawaniem G części tekstowej i wartości zmiennej.

7 Po uruchomieniu programu na wyjściu danych powinny pojawić się już dwie informacje, które w prawidłowy sposób wskazują na największą i najmniejszą wartość w zbiorze H.

```
dane wejściowe   Wyjście
Sukces #stdin #stdout 0.02s 24616KB
Największa z liczb to:765
Najmniejsza z liczb to:26     H
```

C# – GDZIE JESZCZE GO SPOTKAMY

Jeśli chcemy tworzyć aplikacje użytkowe w języku C#, możemy wykorzystać do tego oprogramowanie **Visual Studio 2019 Community** (DVD-KOD: 012), które jest zintegrowanym środowiskiem programistycznym firmy Microsoft. Ten właśnie program jest wykorzystywany w Unity do edycji kodów źródłowych.
Skrypty silnika gier Unity, **ideone.com** czy IDE Microsoftu, nie są jedynymi miejscami, gdzie możemy pisać kody w języku C#. Język ten jest popularny i wykorzystywany także w innych miejscach. Przykładem może być silnik gier **Godot** wspominany już w pierwszym rozdziale **1**.
Język C# nie był w Godocie pierwszym językiem, w jakim można było tworzyć skrypty. Twórcy tego silnika postawili najpierw na własny język GDScript, który nadal może być wykorzystywany.
Historia ta bardzo przypomina to, co w kwestii języków programowania działo się również z Unity. Zanim C# stał się głównym językiem programowania używanym w Unity, wcześniej był nim język Boo, który został usunięty, gdy wydano Unity 5. Innym wspieranym przez Unity językiem było UnityScript, czyli specjalna wersja języka JavaScript, która przestała być wspierana w sierpniu 2017 roku, po wydaniu Unity 2017.1.
Jak widać, C# jest na tyle wdzięcznym językiem programowania, że wielu twórców narzędzi dąży do tego, by z niego korzystać.

Strona projektu Godot Engine



Na DVD znajdziemy plik projektu przedstawionego w tym rozdziale

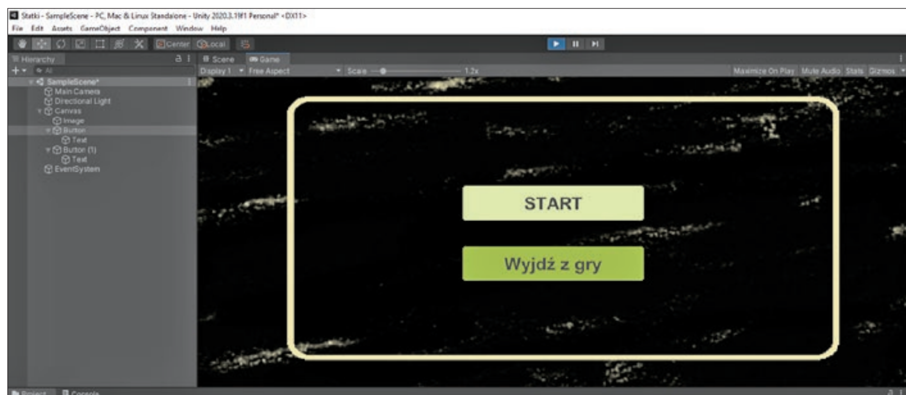
5 Pierwsza gra: Statki kosmiczne

Dysponując wiedzą na temat tego, jak zrobić menu gry, i znając podstawy programowania w języku C#, możemy przejść do tworzenia własnej gry

Rozpoczynamy od menu

Korzystając ze wskazówek z rozdziału **3** tej książki, jesteśmy już w stanie stworzyć własne menu gry.

I od tego etapu rozpoczynamy pracę nad naszą pierwszą grą. Oczywiście gotowe menu powinno komponować się z tematem realizowanej gry.



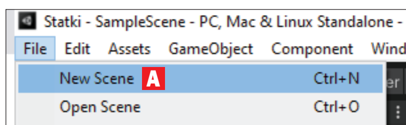
Tworzymy kosmos

Motywem przewodnim gry, której tworzenie jest opisane w tym rozdziale, jest statek kosmiczny. Gra będzie zrealizowana z wykorzystaniem grafiki 3D, z widokiem z góry na znajdujące się na scenie obiekty. Zadaniem gracza będzie sterowanie statkiem kosmicznym i przesuwanie go na boki tak, by uniknąć kolizji ze zmierzającymi w jego stronę obiektami.

Tworzymy nową scenę

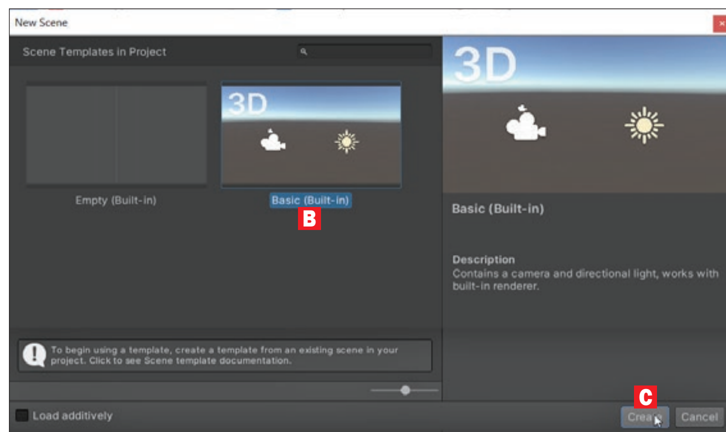
Dysponujemy już projektem zawierającym jedną scenę, czyli menu. Do takiego projektu należy dołożyć kolejną scenę, zawierającą rozgrywkę.

1 By utworzyć nową scenę, rozwijamy pozycję **File** z paska menu górnego.

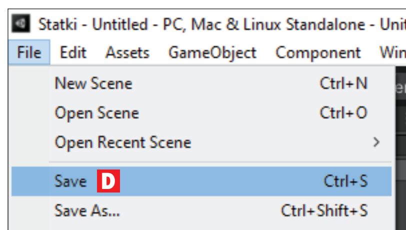


2 Z wyświetlonych opcji wybieramy pierwszą pozycję, czyli **New Scene** **A**.

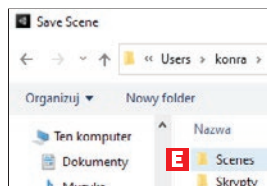
3 W nowym oknie należy wybrać szablon sceny, możemy skorzystać z szablonu **Basic** **B**, który zawiera już kamerę i światło. Po jego zaznaczeniu klikamy na przycisk **Create** **C**, aby scena została utworzona.



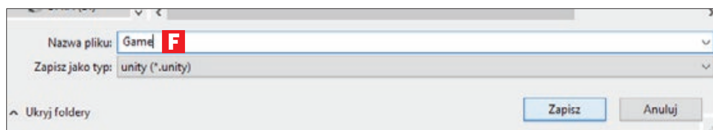
4 Po utworzeniu nowej sceny zapisujemy ją w określonej lokalizacji i pod wybraną nazwą. To ważne, ponieważ za chwilę będziemy programować działanie przycisku, który będzie przełączał gracza na utworzoną właśnie scenę, i będziemy musieli ją wskazać, podając jej nazwę. Ponownie rozwijamy pozycję **File** z paska menu górnego i tym razem wybieramy z listy opcji **Save** **D**.



5 W oknie, jakie zostanie otwarte, należy wybrać lokalizację zapisywanej sceny. Domyślnie wskazywane są zasoby projektu. W nich znajduje się folder **Scenes** **E**, przechodzimy do niego.



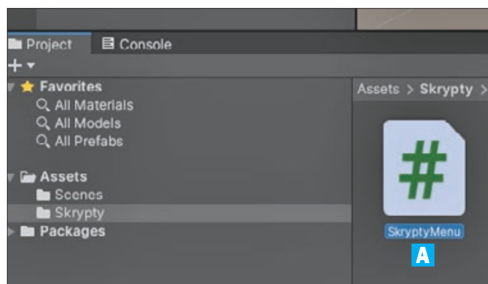
6 Uzupełniamy pole na nazwę pliku, czyli nazwę sceny. Niech nazywa się **Game** **F** (na kolejnej stronie). Klikamy na **Zapisz**.



Przypisujemy działanie do przycisku startującego rozgrywkę

Skoro dysponujemy już sceną, na której będzie znajdowała się cała rozgrywka, możemy zaprogramować niedziałający jeszcze przycisk menu, który ma służyć właśnie do przechodzenia do rozgrywki.

1 Z poziomu zasobów projektu otwieramy plik ze skryptem **A**, w którym zapisana była wcześniej funkcja obsługująca zamykanie gry.



2 Pierwszym elementem, jaki należy dodać do istniejącego skryptu, jest kolejna przestrzeń nazw, **UnityEngine.SceneManagement**, dodamy ją, zapisując pod dotychczas danymi wcześniej przestrzeniami nazw kolejną linię z dyrektywą **using**.

C# więcej na stronie 30

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;
```

3 W skrypcie powinna znaleźć się też funkcja odpowiadająca za rozpoczęcie gry. Dodajemy zatem funkcję typu **void** o nazwie **Zaczynamy** **B**.

4 W nowej funkcji powinno znaleźć się polecenie **SceneManager.LoadScene("Game");** **C**, gdzie **Game** oznacza nazwę sceny, którą chcemy uruchomić.

```
public class SkryptyMenu : MonoBehaviour
{
    public void Zamykanie()
    {
        Application.Quit();
        Debug.Log("Zamykanie...");
    }

    public void Zaczynamy() B
    {
        |
    }
}
```

C# więcej na stronie 41

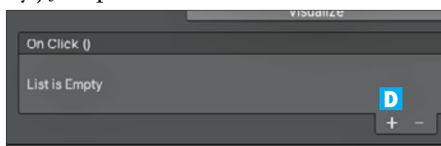
Skorzystanie z tego polecenia jest możliwe dzięki wcześniejszemu dodaniu do projektu kolejnej przestrzeni nazw.

```
public void Zaczynamy()
{                                  C
    SceneManager.LoadScene("Game");
}
```

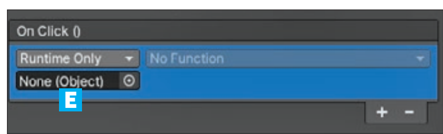
5 Zapisujemy zmieniony skrypt i wracamy do edytora Unity.

6 Przechodzimy do sceny z menu i zaznaczamy na niej przycisk rozpoczynający rozgrywkę, tak by panel właściwości wyświetlał właściwości tego przycisku.

7 Wśród dostępnych właściwości znajdujemy tę, która odpowiada za zdarzenie kliknięcia na przycisk, czyli **On Click**. Jest ona pusta, klikamy na przycisk z plusem **D**, by ją uzupełnić.



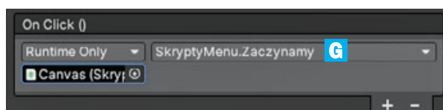
8 W nowo wyświetlonym polu klikamy na **None (Object)** **E** i z nowego okna wy-



bieramy obiekt sceny, do którego przypisany jest plik ze skryptem, czyli, tak jak było to w przypadku przycisku zamykającego grę, obiekt **Canvas**, który znajdziemy w zakładce **Scene**.



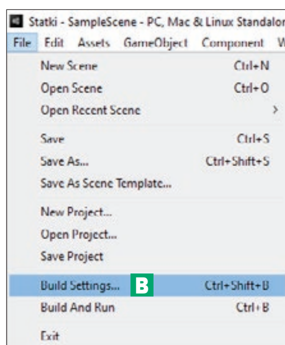
9 W polu **No Function** **F** należy teraz odszukać funkcję, która powinna przeładować scenę na tę zawierającą właściwą rozgrywkę. Powinna to być funkcja **Zaczynamy** **G**.

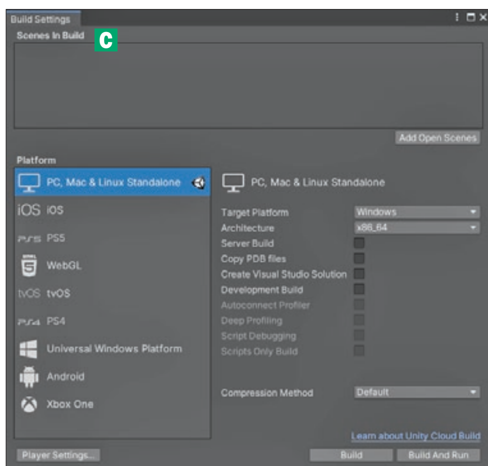


Połączenie scen projektu

1 Gdybyśmy na tym etapie chcieli przetestować działanie naszego skryptu, moglibyśmy zobaczyć błąd **A** na pasku u dołu okna edytora.

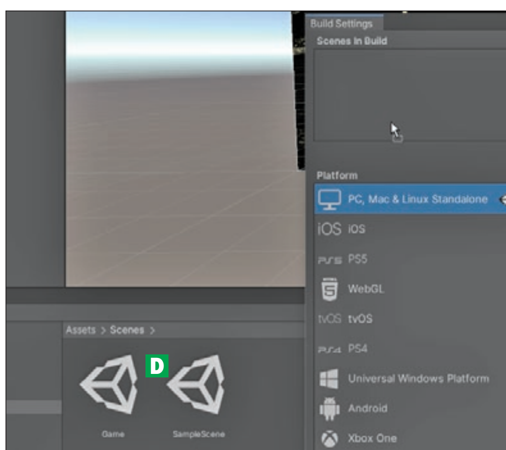
2 Błąd ten mówi o tym, że we właściwościach projektu dotyczących już efektu końcowego naszej gry nie ma jeszcze wskazanych scen, które w efekcie końcowym powinny się znaleźć. Nie ma połączenia między dwoma scenami, mimo że istnieją one w obrębie tego samego projektu. Takie połączenie należy utworzyć. W tym celu kolejny raz rozwijamy z menu górnego pozycję **File** i tym razem z listy opcji wybieramy **Build Settings...** **B**.





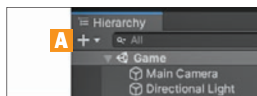
3 Wyświetlone zostanie nowe okno zawierające ustawienia końcowe gotowej gry, jaka ma wyjść z naszego projektu. W oknie tym znajduje się pusta sekcja **Scenes in Build** **C**.

4 Z zasobów projektu w oknie edytora należy przeciągnąć posiadane sceny **D** do sekcji **Scenes in Build** w oknie ustawień, począwszy od sceny z menu.



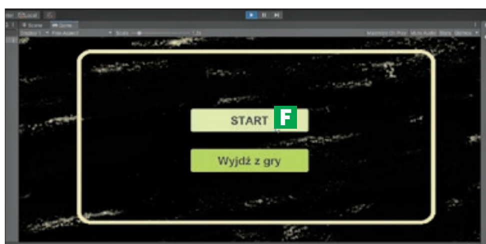






5 Obydwie sceny powinny pojawić się na liście **E** w sekcji **Scenes in Build**, a scena z menu powinna być pierwsza, jako scena startowa projektu.

6 Gdy teraz uruchomimy podgląd gry na scenie z menu, klikamy na **START** **F**.



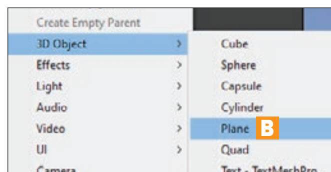
7 Na podglądzie ukaże się zawartość pustej **G** jeszcze sceny **Game**. Naszym dalszym zadaniem będzie jej zapełnienie, tak by pojawiła się tam właściwa rozgrywka.

Tworzymy kosmos

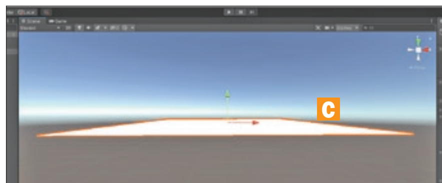
Nasza gra rozgrywa się w przestrzeni kosmicznej. W widoku z góry na obiekty pod nimi powinno być widoczne tło przedstawiające kosmos. Właśnie od dodania tego tła rozpoczniemy tworzenie obiektów na scenie.

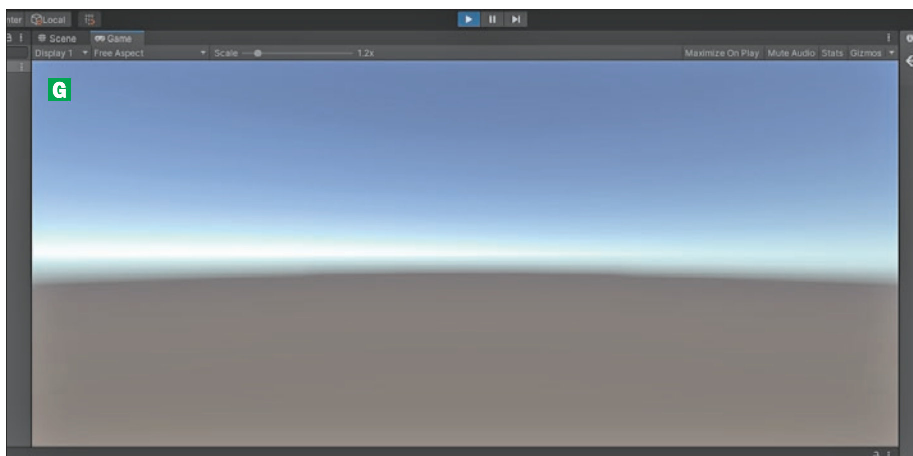
1 Dodajemy płaszczyznę, która będzie pełniła rolę tła w grze. Klikamy na przycisk z plusem w panelu **Hierarchy** **A**.

2 Następnie z rozwiniętej listy kategorii możliwych do dodania obiektów wybieramy kategorię **3D Object**, a następnie obiekt **Plane** **B**.



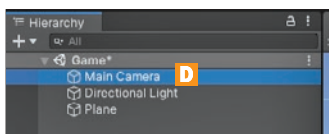
3 Na scenie pojawi się płaszczyzna **C**. Ma ona pełnić rolę tła naszej gry. Jeśli teraz



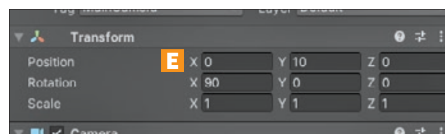


przejdziemy na zakładkę **Game** lub uruchomimy podgląd sceny, zobaczymy, że widok z kamery nie pokazuje naszej płaszczyzny. Możemy jednak odpowiednio przestawić kamerę, by znajdowała się nad płaszczyzną i pokazywała ją z góry, dzięki temu płaszczyzna będzie tłem dla naszej rozgrywki. Musimy dobrać odpowiednie położenie zarówno płaszczyzny, jak i kamery.

4 W spisie obiektów na scenie zaznaczamy obiekt **Main Camera** **D**, by w inspektorze wyświetlone zostały jego właściwości.



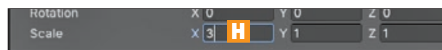
5 W sekcji **Transform** mówiącej o położeniu i ułożeniu obiektu należy ustawić **Position** na **x: 0, y: 10, z: 0** **E**. Z kolei **Rotation** trzeba ustawić na **x: 90, y: 0, z: 0**, dzięki temu kamera obróci się w taki sposób, że widok z niej nie będzie trafiał na bok płaszczyzny.



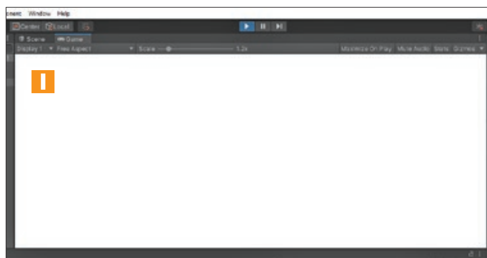
6 Również płaszczyzna powinna mieć dobrane położenie i wymiary w taki sposób, by znalazła się w obszarze obejmowanym przez kamerę. Zaznaczamy płaszczyznę, by wyświetlić jej właściwości, i ustawiamy w nich **Position** na **x: 0, y:0, z:0** **F**. Również **Rotation** należy ustawić na **x: 0, y:0, z:0** **G**.



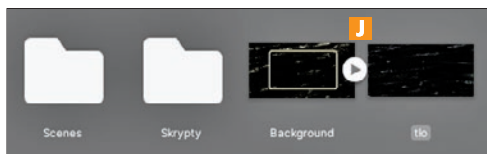
7 Jeśli teraz uruchomimy podgląd sceny, zobaczymy, że widok z kamery obejmuje już płaszczyznę. Kamera wciąż pokazuje jednak więcej niż tylko samą płaszczyznę. By temu zaradzić, możemy przeskalować płaszczyznę tak, by ją powiększyć. W jej właściwościach, w sekcji **Transform** zmieniamy skalę na osi X z 1 na **3** **H**.



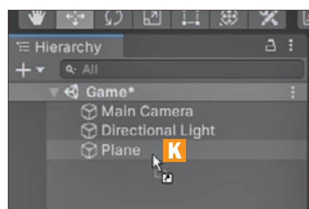
8 Po uruchomieniu podglądu zobaczymy, że płaszczyzna wypełnia cały widok z kamery **I**.

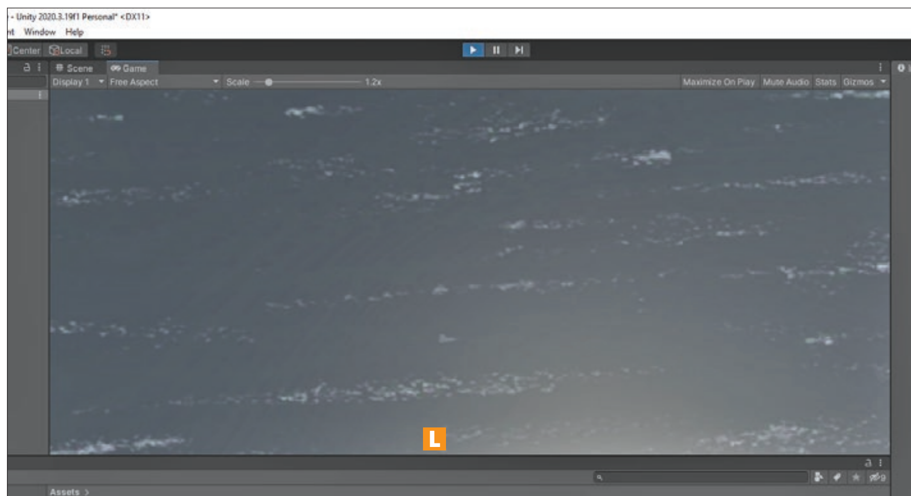


9 Płaszczyzna ma jeszcze swój domyślny kolor. Możemy go zmienić, pokrywając ją materiałem. W ten sposób nadaje się kolory i tekstury obiektom. Materiały można tworzyć samodzielnie jako nowe zasoby projektu, można też skorzystać opcji automatycznego tworzenia materiału na podstawie tekstury. W tym celu dodajemy do zasobów projektu plik graficzny, jaki chcemy zobaczyć w tle gry **J**.



10 Plik graficzny z zasobów projektu przeciągamy na płaszczyznę **K**.



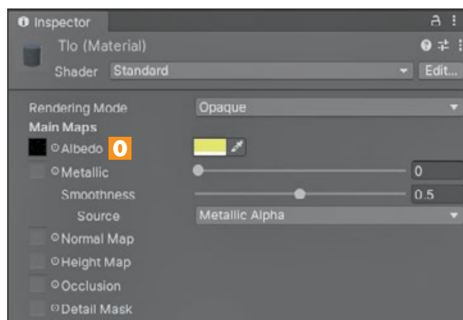


11 Płaszczyzna pokryje się już materiałem **L** stworzonym na podstawie pliku graficznego.



12 Zwróćmy też uwagę, że jednocześnie w zasobach projektu pojawił się folder **Materials** **M**.

13 W tym folderze znajduje się utworzony automatycznie materiał **N**, którego właściwości można zmieniać poprzez sekcję **Inspector**. Na przykład, by nadać materiałowi nieco inny odcień, zmieniamy jego **Albedo** **O**, wybierając dowolny kolor.



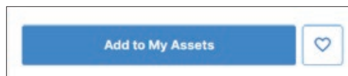
Asset Store

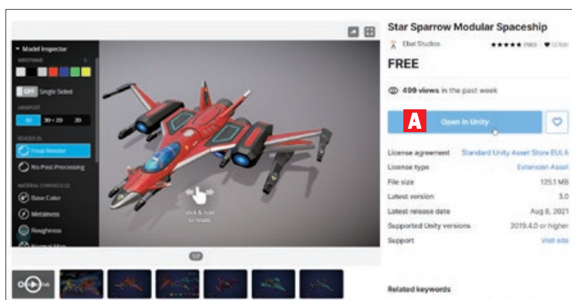
We wcześniejszych wersjach Unity integralną częścią edytora był **Asset Store**, czyli sklep z zasobami, jakie można wykorzystać w swoich projektach. Obecnie Asset Store nadal istnieje, jednak w formie odrębnej strony internetowej pod adresem **assetstore.unity.com**. W sklepie tym znajdziemy zarówno zasoby płatne, jak i darmowe. By z nich skorzystać, należy zalogować się do sklepu poprzez to samo konto, poprzez które logujemy się do edytora Unity i aplikacji Unity Hub. W tym rozdziale wykorzystamy Asset Store do pobrania i umieszczenia w naszym projekcie modeli 3D. Jeden z nich będzie przedstawiał statek kosmiczny, a drugi asteroidę.

Dodajemy statek kosmiczny

1 Znajdujemy w Asset Store zasób o nazwie **Star Sparrow Modular Spaceship**.

2 Gdy odszukamy zasób, klikamy na przycisk **Add to My Assets**.

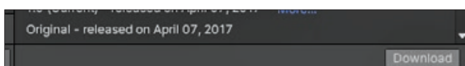




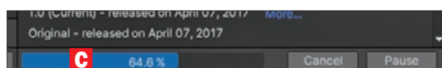
3 Po dodaniu obiektu do naszych zasobów przycisk, na który kliknęliśmy, zmienia się w przycisk **Open in Unity** A. Klikamy na niego.

4 Zostajemy przeniesieni do edytora Unity, w którym otwiera się okno **Package Manager** B, gdzie znajdziemy zasób wybrany przez nas w Asset Store.

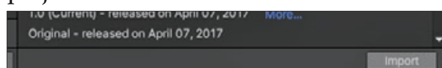
5 By móc skorzystać z tego zasobu, należy go pobrać. Klikamy na przycisk **Download** znajdujący się w prawym dolnym rogu okna Package Managera.



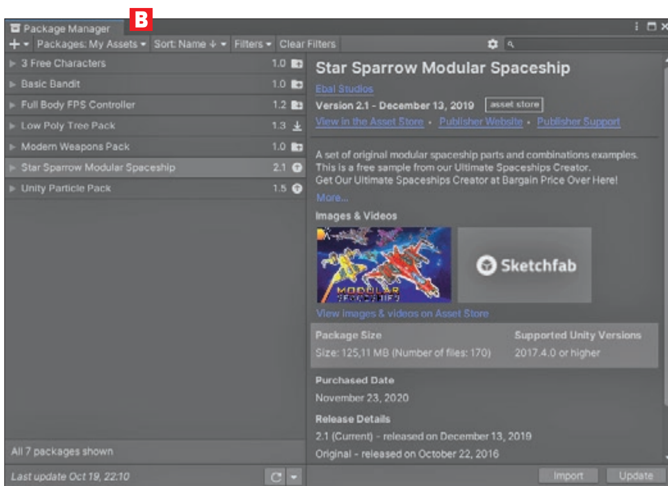
6 Pobieranie rozpocznie się, a my możemy obserwować jego postępy na pasku C.

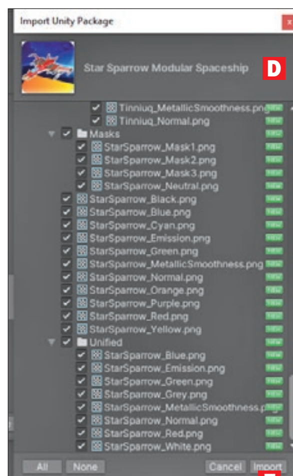


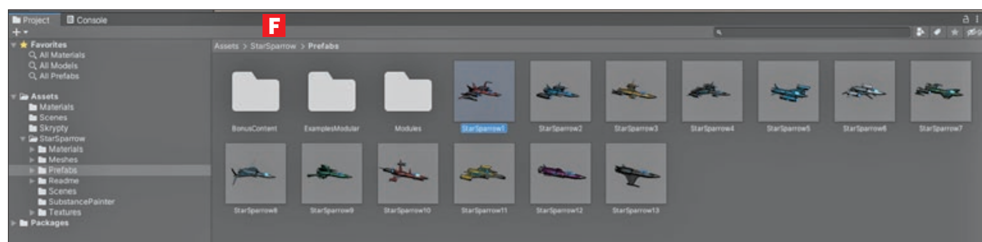
7 Gdy pobieranie się zakończy, widoczny staje się przycisk **Import**, na który należy kliknąć, aby móc dodać pobrany zasób do projektu.



8 Następnie otwarte zostaje kolejne okno. W nim znajduje się lista D wszystkich plików, jakie budują wybrany przez nas zasób. Domyślnie wszystkie pliki są na niej zaznaczone, nie musimy tego zmieniać, choć nie wszystkie elementy zasobu będą potrzebne w naszym projekcie. Jeśli klikniemy na przycisk **Import** E, całość zasobu zostanie dodana do projektu.





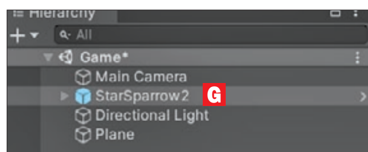


9 W przypadku opisywanej paczki statków kosmicznych po zakończeniu importu w zasobach projektu pojawi się nowy folder, **StarSparrow** **F**. Zawiera on kilka podfolderów, między innymi **Prefabs**. W nim znajdują się prefabrykaty, czyli gotowe obiekty zawierające nie tylko trójwymiarowe modele, ale też zestawy komponentów pozwalające na wykorzystanie takiego obiektu w grze. W przypadku tej paczki zasobów w folderze tym znajdziemy zestaw różnych statków kosmicznych.

10 Wybieramy jeden ze statków i przeciągamy na panel modelowania sceny.



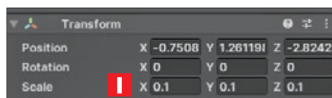
11 W panelu **Hierarchy** zobaczymy, że nowy obiekt **G** pojawił się na liście obiektów sceny.

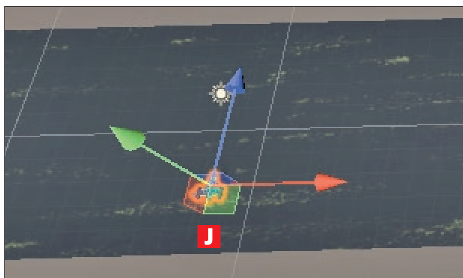


12 Jeśli uruchomimy teraz podgląd sceny, zobaczymy, że wybrany statek kosmiczny może być za duży **H** do wykorzystania go w takiej formie podczas rozgrywki.



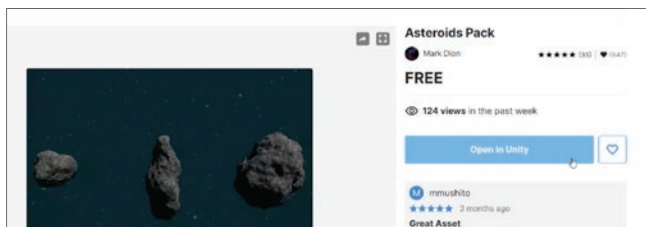
13 W takiej sytuacji należy zmniejszyć obiekt. Można to robić z poziomu panelu właściwości, czyli panelu **Inspector** po prawej stronie okna edytora. W grupie właściwości **Transform** znajduje się właściwość **Scale**. W niej wszystkie pola mają domyślnie wartość 1. Aby zmniejszyć obiekt, należy wpisać tam wartość mniejszą niż 1. Dobieramy ją tak, by wielkość statku kosmicznego odpowiadała naszym oczekiwaniom. Dobrą wartością we wszystkich polach może być **0.1** **I**. Po jej wpisaniu statek wyraźnie się zmniejszy **J**.

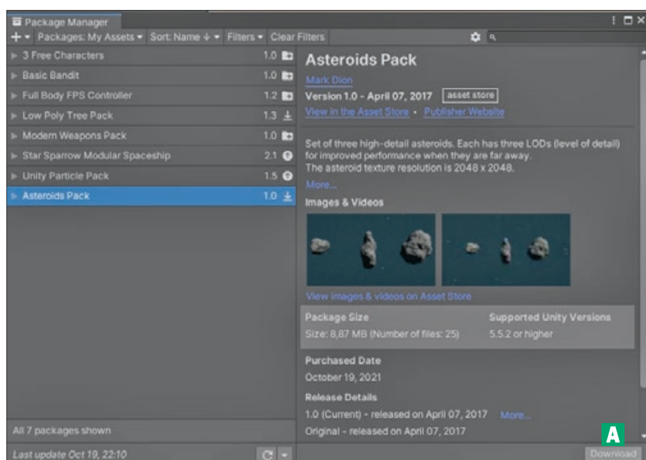




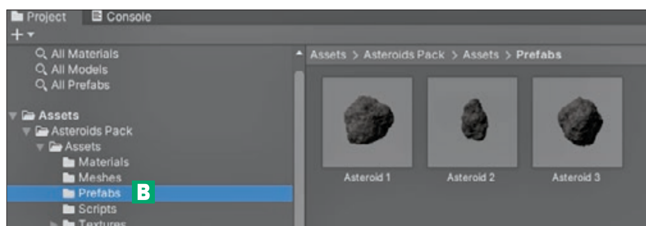
Dodajemy do projektu asteroidę

1 W przypadku dodawania asteroidy do projektu możemy postąpić podobnie jak w przypadku statku kosmicznego. Najpierw znajdujemy w Asset Store zasób o nazwie **Asteroid Pack**, a następnie wykonujemy te same kroki co w przypadku statku kosmicznego, by dodać zasób do projektu. Gdy zasób będzie widoczny w oknie Package Managera, pobieramy go poprzez przycisk **Download** **A** i importujemy do zasobów projektu.

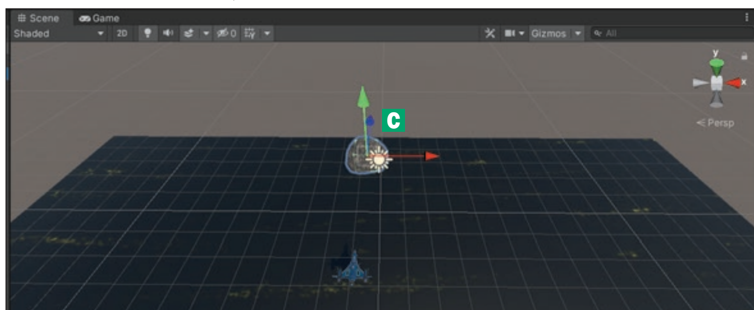




2 Paczka asteroid w zasobach projektu będzie znajdowała się w folderze **Asteroid Pack**. Są w nim dwa foldery, **Demo** i **Assets**, otwieramy drugi z nich, czyli **Assets**. W nim jest katalog **Prefabs** **B**, czyli podobnie jak w przypadku statków kosmicznych, mamy do czynienia z prefabrykatami. Są to gotowe obiekty zawierające już pewne ustawienia, asteroidy w tej paczce mają nawet domyślnie podłączony skrypt, który sprawia, że się obracają.



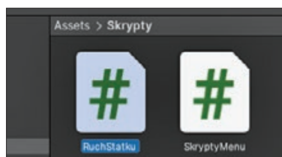
3 Po dodaniu takiego obiektu do projektu i przeciągnięciu go na scenę, gdy uruchomimy podgląd, obiekt ten będzie już w ruchu, nie musimy go sami programować. Przeciągamy zatem wybraną asteroidę na scenę **C**, uruchamiamy podgląd i sprawdzamy, jak wygląda jej ruch. Umieszczając asteroidę na scenie, pamiętajmy o tym, by przeciągnąć ją w takie miejsce, by była ona w obrębie widoku z kamery.



Programujemy sterowanie statkiem

Mimo że paczka zasobów ze statkami kosmicznymi jest dość bogata, to znajdujące się w niej statki i tak należy nieco rozbudować, by móc je dobrze wykorzystać w grze. Rozbudowanie będzie polegać na dodaniu do nich kolejnych możliwości, czyli komponentów. Niektóre z komponentów są zdefiniowane przez silnik Unity i samo ich dołączenie do obiektu może zapewnić pożądane działanie, a niektóre należy napisać samodzielnie, tworząc odpowiedni skrypt. Zrobimy tak, tworząc skrypt sterowania statkiem kosmicznym.

1 Jak tworzyć nowe skrypty w zasobach projektu, wiemy już z rozdziału dotyczącego tworzenia menu. Tam stworzyliśmy skrypt, który obsługuje kliknięcia na przyciski menu. Tym razem stworzymy skrypt, który dołączymy do statku kosmicznego znajdującego się na scenie. W folderze **Skrypty** w zasobach projektu tworzymy skrypt o nazwie **RuchStatku**.



2 Struktura nowego skryptu jest taka, jaka była w przypadku tworzenia skryptu dotyczącego menu. Zawiera ona dwie funkcje – **Start** i **Update**. Tym razem nie będziemy ich usuwać, ale zajmiemy się edycją funkcji **Update**.

```
void Update()
{

}
```

3 To, co w niej zapiszemy, będzie wykonywane przez program wraz z wyświetleniem każdej kolejnej klatki gry na ekranie. Można powiedzieć, że treść tej funkcji wykonuje się cały czas, gdy obiekt jest na scenie. Jeśli nasz statek kosmiczny cały czas ma mieć możliwość ruchu, to właśnie w tym skrypcie zapiszemy instrukcje pozwalające na ten ruch. Żeby to zrobić, musimy odpowiedzieć sobie na pytanie, w jaki sposób chcemy sterować naszym statkiem. Jeśli ma się to odbywać poprzez wciskanie odpowiednich klawiszy na klawiaturze, to w funkcji tej powinniśmy sprawdzać, czy wciśnięto jakiś klawisz, a także, jaki to był klawisz. W zależności od wyniku sprawdzania statek będzie się przemieszczać w odpowiednią stronę. Statek, by unikać kolizji z asteroidą, powinien móc przesuwać się zarówno w prawą, jak i lewą stronę, i taki ruch statku zapiszemy w skrypcie.

4 Pierwszym elementem funkcji powinna być instrukcja warunkowa **if**.

```
void Update()
{
    if ()
    {

    }
}
```

C# więcej na stronie 34

5 W niej powinniśmy najpierw sprawdzić, czy w ogóle wciśnięto jakiś klawisz, zapisujemy więc w miejscu na warunek **Input.GetKey()** A.

```
if (Input.GetKey())
{

}
```

A

6 Jeśli wciśnięto klawisz, to musimy sprawdzić, jaki był to klawisz. Rozpoczniemy od sprawdzenia wciśnięcia klawisza strzałki w prawą stronę. W nawiasie po **GetKey** zapisujemy **KeyCode.RightArrow** B, co w Unity jest odniesieniem do tego klawisza.

B

```
if (Input.GetKey(KeyCode.RightArrow))
```

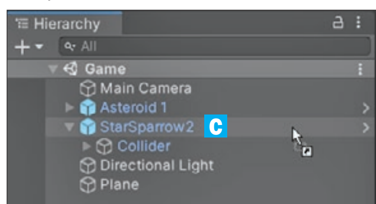
7 Zanim zapiszemy w skrypcie odpowiednią instrukcję, która faktycznie przesunie nasz statek kosmiczny w prawą stronę, zapiszmy wewnątrz nawiasu klamrowego,

jako instrukcję do wykonania po spełnieniu warunku, polecenie **Debug.Log** z informacją o tym, co powinien zrobić statek.

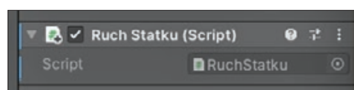
```
{
    Debug.Log("Ruch w prawą stronę");
}
```

8 Zapisujemy skrypt i wracamy do edytora Unity.

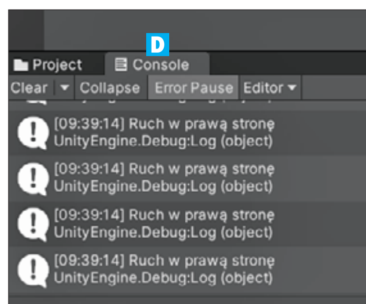
9 Przystąpimy teraz do połączenia skryptu z obiektem i sprawdzenia, czy zapisana przez nas instrukcja warunkowa poprawnie wykrywa wciśnięcie strzałki w prawą stronę. Przeciągamy z zasobów projektu plik z tworzonym teraz skryptem na obiekt, statek kosmiczny **C**.



10 Upewniamy się co do prawidłowego przyłączenia skryptu do obiektu, w panelu właściwości statku kosmicznego powinna pojawić się sekcja odpowiadająca za ruch statku.



11 Uruchamiamy podgląd rozgrywki i naciskamy klawisz strzałki w prawą stronę. W zakładce **Console** **D**, do której



możemy się przełączyć na panelu z zasobami projektu, powinny się pojawiać wpisy z polecenia **Debug.Log**.

12 Nawet w wyniku krótkiego wciśnięcia klawisza wpis może pojawić się kilkakrotnie. Gdybyśmy trzymali klawisz wciśnięty przez jedną sekundę, wpis pojawiłby się tyle razy, ile klatek rozgrywki byłoby wyświetlonych w ciągu tej sekundy.
Liczba klatek na sekundę to tak zwana wartość fps (ang. frames per second), może być ona różna w zależności od sprzętu, na którym uruchamiamy naszą produkcję. Mocniejszy sprzęt jest w stanie wykonać więcej klatek na sekundę niż słabszy.
Musimy się zastanowić, jakie ma to znaczenie dla tworzonych przez nas skryptów, a ma kluczowe.
Jeśli z każdym wykonaniem funkcji przesuwalibyśmy statek kosmiczny o tę samą odległość, to w przypadku większej liczby klatek na sekundę przesunięcie wykonałoby się więcej razy niż w przypadku mniejszej liczby klatek na sekundę.
To znaczy, że na dwóch różnych sprzętach wciśnięcie klawisza trwające tyle samo czasu skutkowałoby tym, że ostatecznie statek przesunie się na każdym z nich o inną odległość.
Nie jest to rozwiązanie dobre, jednak możemy temu zaradzić, tak dobierając odległość przesunięcia z każdym wywołaniem funkcji, aby była uzależniona od tego, ile czasu minęło od poprzedniego jej wywołania. W ten sposób często wywołująca się funkcja będzie przesuwała statek o mniejszą odległość z każdym wywołaniem niż funkcja rzadziej wywoływana.

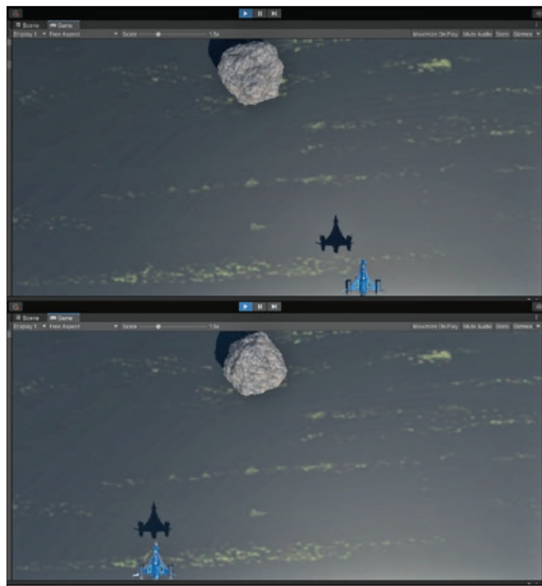
13 Wracamy do skryptu. Będziemy go teraz edytować. Ruch obiektu to tak naprawdę zmiana jego pozycji, do której poprzez skrypt odnosimy się jako **transform.position** **E** (patrz kolejna strona). Naszym zadaniem jest taka zmiana tej pozycji, by obiekt przesunął się w prawą stronę. Pozycja w trójwymiarowym świecie opisywana jest za pomocą trzech współrzędnych: **x**, **y**, **z**.

```
    Debug.Log("Ruch w prawą stronę");                      F
E   transform.position = transform.position + Vector3.right;
}
```

```
                                                                   G
transform.position = transform.position + Vector3.right * Time.deltaTime;
```

Unity daje nam mechanizm, dzięki któremu nie musimy nawet bezpośrednio wskazywać, którą z tych trzech wartości chcemy zmienić. Wystarczy, że do obecnej pozycji dodamy trójwymiarowy wektor przesunięcia w prawą stronę, który w Unity zapisujemy jako **Vector3.right**. Zatem nowa pozycja obiektu po wciśnięciu strzałki to stara pozycja, do której dodajemy wektor, co zapisujemy jako: **transform.position = transform.position + Vector3.right;** F.

14 I na tym etapie zaprogramowaliśmy to właśnie w taki sposób, że z każdą klatką rozgrywki przesunięcie jest stałe i niezależne od liczby klatek na sekundę. Przed dodaniem całego wektora przesunięcia moglibyśmy pomnożyć go przez wartość **Time.deltaTime** G, czyli czas, jaki minął od poprzedniej klatki. Im ten czas jest mniejszy, tym mniejszy będzie wynik mnożenia, a co za tym idzie, odległość przesunięcia statku.


Statek kosmiczny przesunięty za pomocą klawiszy strzałek w czasie trwania rozgrywki

```
void Update()
{
    if (Input.GetKey(KeyCode.RightArrow))
    {
        Debug.Log("Ruch w prawą stronę");
        transform.position = transform.position + Vector3.right * Time.deltaTime
    }
    if (Input.GetKey(KeyCode.LeftArrow)) H
    {

    }
```

```
if (Input.GetKey(KeyCode.LeftArrow))
{         I
    Debug.Log("Ruch w lewą stronę");                  J
    transform.position = transform.position + Vector3.left * Time.deltaTime;
}
```

15 Żeby nasz statek mógł poruszać się też w lewą stronę, w **Update** powinna pojawić sie kolejna instrukcja warunkowa, sprawdzająca, czy wcisnęliśmy klawisz strzałki w lewą stronę. Jej warunkiem będzie zatem **Input.GetKey(KeyCode.LeftArrow)** H. informację w konsoli, a potem dodać zmianę pozycji poprzez dodanie do niej wektora przesunięcia pomnożonego przez czas, jaki upłynął od ostatniego odświeżenia klatki. W przypadku przesunięcia w lewą stronę należy dodać do pozycji wektor **Vector3.left** J.

16 Instrukcje, jakie mają się wykonać po wciśnięciu tego klawisza, są podobne do tego, co dzieje się w przypadku strzałki w prawą stronę. Najpierw możemy skorzystać z **Debug.Log** I, by wyświetlić

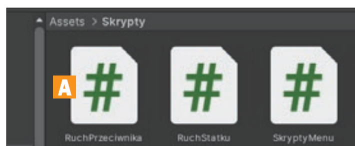
17 Po takich zmianach statek powinien poruszać się na boki we właściwy sposób, a jego prędkość nie będzie zależna od możliwości sprzętu, na jakim uruchamiamy naszą grę.

Programujemy asteroidę

Ruch asteroidy

Nasz statek potrafi się już poruszać na boki, natomiast asteroida, poza domyślnie dołączonym do niej skryptem powodującym jej ruch obrotowy, jeszcze się nie przemieszcza. W tej części rozdziału zajmiemy się właśnie zaprogramowaniem tego, by asteroida się przemieszczała.

1 Jeśli asteroida ma się przemieszczać, to znaczy, że tak jak statek kosmiczny powinna wykonywać pewien skrypt. Utwórzmy więc skrypt o nazwie **RuchPrzeciwnika** A i dołączmy go do asteroidy, przeciągając plik z zasobów na obiekt.



2 Otwieramy plik ze skryptem. Wprowadzimy do niego zmiany sprawiające, że asteroida będzie poruszała się w dół okna gry, czyli w stronę naszego statku kosmicznego. Zadaniem gracza będzie natomiast takie przesuwanie statku, by uniknąć kolizji z nadlatującą asteroidą. Sam ruch asteroidy będzie czymś podobnym do ruchu statku kosmicznego, czyli zmianą pozycji opisanej jako **transform.position** przez dodanie do niej odpowiedniego wektora. Ruch ten powinien być wykonywany z każdą klatką rozgrywki, zatem powinniśmy umieścić go w funkcji **Update**. W przeciwieństwie do statku kosmicznego, tym razem nie musimy określać ruchu w dwóch kierunkach i sprawdzać, czy wciśnięto jakiś klawisz. Ruch ten ma być wykonywany niezależnie od zachowań gracza, zatem zmianę pozycji możemy zapisać bezpośrednio w funkcji. By asteroida podążała w dół okna gry, należy do jej pozycji dodawać wektor **Vector3.back** B, warto pamiętać również o tym, by wartość wektora pomnożyć jeszcze przez czas od ostatniego odświeżenia klatki, co uniezależni prędkość asteroidy od możliwości sprzętowych.

3 Gdy zapiszemy tak zmieniony skrypt i uruchomimy podgląd, zobaczymy, że

```
void Update()
{
    transform.position = transform.position + Vector3.back * Time.deltaTime;   // B
}
```

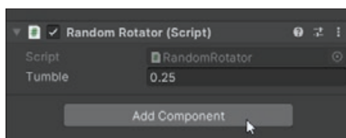


asteroida przemieszcza się już w kierunku statku kosmicznego.

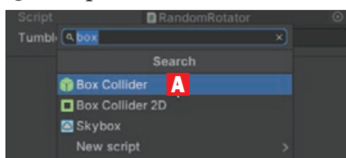
Wykrywanie kolizji ze statkiem

Lecąca w stronę statku asteroida może w niego uderzyć, jednak na tym etapie pracy nic się nie stanie, nie zobaczymy żadnego efektu kolizji między tymi obiektami. Musimy to dopiero zaprogramować. W tej części rozdziału zajmiemy się wykrywaniem kolizji, czyli uderzenia asteroidy w statek kosmiczny.

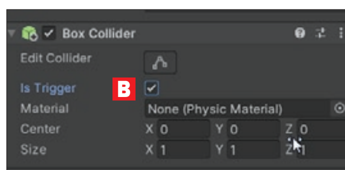
1 By obiekt mógł wykrywać kolizję z innym obiektem, musi mieć dołączony **Collider**, czyli komponent na to pozwalający. Komponentów z wyrazem Collider w nazwie jest kilka. Mogą one dawać obszary kolizji o różnych kształtach. Na potrzeby naszej asteroidy wybieramy **Box Collider**. By dodać go do obiektu, przechodzimy do właściwości asteroidy i u dołu panelu z właściwościami klikamy na przycisk **Add Component**.



2 W polu wyszukiwania podajemy nazwę poszukiwanego komponentu i klikamy na znalezioną pozycję **Box Collider** **A**. Po dodaniu tego komponentu asteroida będzie reagować na kolizję ze statkiem kosmicznym, który stanie się przeszkodą na trasie. Silnik pozwoli nam na fizyczne odwzorowanie oporu, jaki postawi statek asteroidzie w jej ruchu. Ta nawet delikatnie zmieni swój tor ruchu, zderzając się ze statkiem. Nam jednak chodzi o to, by kolizja była wyłapana z poziomu skryptu, dlatego posłużymy się **Triggerem**, czyli, inaczej mówiąc, **wyzwalaczem**. Jest to coś w rodzaju fotokomórki, która wykryje fakt nachodzenia na siebie dwóch obiektów, ale nie będzie to wpływało na ich faktyczny ruch.



3 By skorzystać z wyzwalacza we właściwościach asteroidy, zaznaczamy w niej opcję **Is Trigger** **B**.



4 Te same kroki, czyli dodanie komponentu **Box Collider** i zaznaczenie w nim opcji **Is Trigger**, należy powtórzyć także dla statku kosmicznego.

5 Następnie należy przejść do skryptu i w nim stworzyć nową funkcję. Musi mieć ona konkretną nazwę, czyli **OnTriggerEnter** **C**, tak aby Unity wiedziało, że tę funkcję należy połączyć właśnie ze zdarzeniem zderzenia dwóch obiektów. Dodatkowo funkcja ta powinna przyjmować parametr typu **Collider**, będzie nim obszar kolizji obiektu, z którym się zderza asteroida. W obrębie funkcji nadamy temu parametrowi nazwę **other** **D**.

```
void OnTriggerEnter(Collider other)
{              C                D

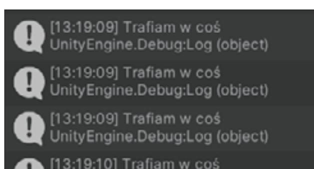
}
```

6 Cokolwiek wpiszemy teraz we wnętrzu tej funkcji, powinno wykonać się w momencie zderzenia asteroidy z jakimś innym

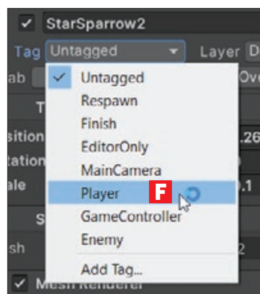
obiektem na scenie. Można teraz umieścić w kodzie polecenie **Debug.Log** E, które wyświetli informację o kolizji.

```
void OnTriggerEnter(Collider other)
{
  E Debug.Log("Trafiam w coś");
}
```

7 Możemy przejść do edytora Unity i uruchomić podgląd rozgrywki. Gdy asteroida uderzy w statek kosmiczny, w oknie konsoli pojawi się odpowiednia informacja.



8 Gdyby na scenie pojawiły się kolejne obiekty, w oknie konsoli widzielibyśmy tę samą informację. Docelowo nasza gra powinna jednak reagować tylko na kolizję między asteroidą a statkiem kosmicznym, a na zderzenie asteroidy z innym obiektem, jeśli taki się pojawi, już niekoniecznie. Dlatego należałoby sprawdzić w skrypcie, z jakim obiektem koliduje asteroida.
Unity stosuje własny system nazw, tagów dla obiektów. Możemy przypisać do obiektu odpowiedni tag, a następnie sprawdzić, jaki jest tag obiektu, z którym koliduje asteroida. Jeśli jest on taki sam jak tag statku kosmicznego, oznacza to kolizję ze statkiem kosmicznym.
By nadać tag statkowi, przechodzimy do panelu jego właściwości. W nim, u samej góry, tuż pod nazwą obiektu znajduje się pole **Tag**, w którym widzimy wartość **Untagged**. Klikamy na nią, by wyświetlić listę tagów dostępnych do użycia. Znajdziemy na niej tag **Player** F, którego zwykle używa się do oznaczenia obiektów sterowanych przez gracza.



9 Wróćmy do skryptu. W nim dodajemy instrukcję warunkową. Sprawdzimy w niej, czy tag obiektu, z którym koliduje asteroida (**other.tag** G), to **Player**.

```
void OnTriggerEnter(Collider other)
{
    Debug.Log("Trafiam w coś");
    if (other.tag == "Player")
    {       G

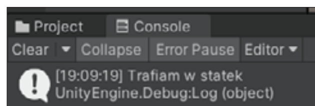
    }
}
```

10 Gdy warunek będzie spełniony, możemy dodać kolejne polecenie **Debug.Log** H wypisujące w konsoli informację o tym, że asteroida trafiła w statek. W ten sposób będziemy mogli zweryfikować poprawność wykrywania kolizji.

```
if (other.tag == "Player")
{
  H Debug.Log("Trafiam w statek");
}
```

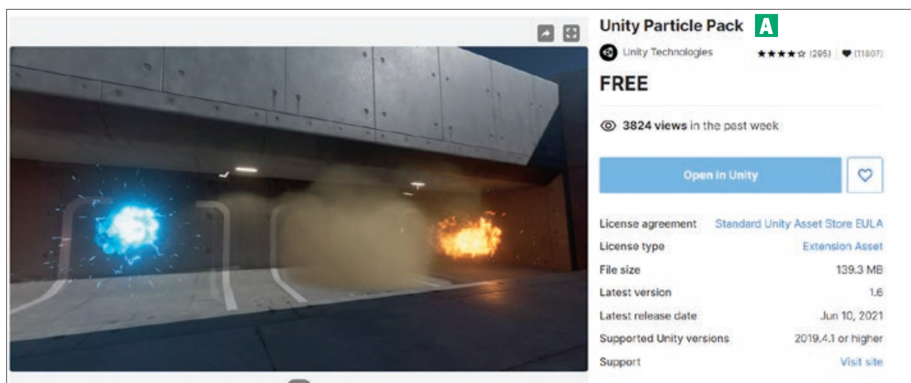
C# więcej na stronie 43

11 Jeśli przetestujemy rozgrywkę, w momencie uderzenia asteroidy w statek kosmiczny zobaczymy w oknie konsoli informację o kolizji ze statkiem.



Wybuch statku kosmicznego

Informacje wyświetlane w konsoli są widoczne jedynie dla nas, w trybie tworzenia gry. Nie są widoczne dla gracza. Poza tym, sama informacja tekstowa to nieco zbyt mało, aby gracz był usatysfakcjonowany takim zakończeniem gry, szczególnie biorąc pod uwagę to, że pracujemy w dość rozbudowanym silniku gier, pozwalającym na osiągnięcie ciekawych efektów graficznych. Dlatego możemy



i warto tak przerobić naszą rozgrywkę, aby w momencie zderzenia asteroidy ze statkiem kosmicznym zobaczyć efektowny wybuch.

1 Pierwszy krok w stworzeniu takiego wybuchu to pobranie i import do projektu odpowiedniej paczki zasobów, w jakiej znajdziemy obiekt, gotowy wybuch do wyświetlenia. Takim zasobem może być **Unity Particle Pack** A.

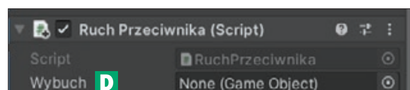
2 Dalej należy wskazać, który z wybuchów i kiedy ma pojawić się na scenie. Ponieważ moment uruchomienia wybuchu, czyli moment kolizji asteroidy ze statkiem, jest wykrywany w skrypcie ruchu asteroidy, również tam będziemy musieli umieścić wszystko to, co dotyczy wybuchu. Jeśli mamy utworzyć na scenie obiekt, to w skrypcie powinno pojawić się jakieś odniesienie do prefabrykatu obiektu, który chcemy umieścić na scenie. Takie odniesienie powinno być dostępne w obrębie całego skryptu. By tak było, trzeba je zadeklarować poza definicją jakiejkolwiek funkcji w skrypcie. Jeżeli chcemy zadeklarować zmienną (a właściwie pole w klasie), w której przechowamy cały obiekt, powinniśmy użyć typu **GameObject**. Nazwa tego elementu w skrypcie to może być **wybuch** B.

```
public class RuchPrzeciwnika : MonoBehaviour
{
    public GameObject wybuch; B
```

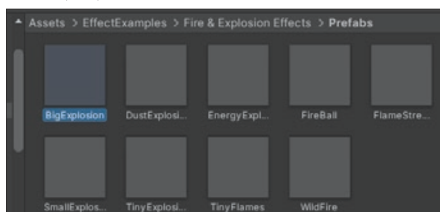
3 Jak teraz połączyć odniesienie do obiektu w skrypcie z prefabrykatem? Możemy to zrobić z poziomu edytora Unity, jednak najpierw powinniśmy przed deklaracją obiektu w skrypcie dodać oznaczenie **[SerializeField]** C.

```
public class RuchPrzeciwnika : MonoBehaviour
{
  C [SerializeField] public GameObject wybuch;
```

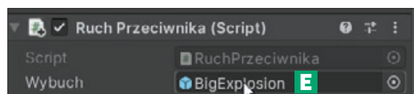
4 Po jego dodaniu we właściwościach asteroidy, gdzie mamy dołączony edytowany właśnie skrypt, pojawi się pole **Wybuch** D, a obok niego informacja o tym, że nie ma tam jeszcze żadnego obiektu (**None**).



5 W zasobach projektu należy odnaleźć prefabrykaty wybuchów. Zasoby z importowanej ostatnio paczki znajdują się w folderze **EffectExamples** zasobów projektu. Same eksplozje są zaś dalej w folderze **Fire & Explosion Effects**, a gotowe prefabrykaty w znajdującym się tam folderze **Prefabs**.



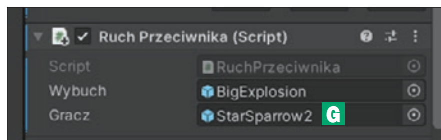
6 Wybieramy jedną z eksplozji (na przykład **BigExplosion**) i przeciągamy ją z zasobów projektu na pole **Wybuch** **E** we właściwościach asteroidy.



7 Obiekt z wybuchem powinien pojawić się na scenie w miejscu, w którym dojdzie do kolizji, czyli właściwie w miejscu, w jakim znajduje się statek kosmiczny, a sam statek powinien wtedy zniknąć. By było to możliwe, musimy w skrypcie asteroidy mieć odniesienie nie tylko do prefabrykatu wybuchu, ale też do obiektu gracza. Zadeklarujmy zatem kolejne pole, do którego będzie można przeciągnąć w edytorze Unity obiekt, czyli statek kosmiczny. Zapisujemy w skrypcie **[SerializeField] public GameObject gracz;** **F**.

F
```
[SerializeField] public GameObject wybuch;
[SerializeField] public GameObject gracz;
```

8 Do nowego pola we właściwościach asteroidy przeciągamy statek kosmiczny (**StarSparrow2** **G**) z panelu **Hierarchy**.



9 Mamy już w naszym skrypcie odniesienie do statku kosmicznego, w momencie kolizji z asteroidą możemy w jego miejscu umieścić wybuch. By utworzyć obiekt na scenie, korzystamy z polecenia **Instantiate**, które w tym konkretnym przypadku może mieć postać: **Instantiate(wybuch, gracz.trans-**

ALTERNATYWNE ODNIESIENIE DO OBIEKTU

Programowanie jest o tyle wdzięcznym zajęciem, że jeden efekt można uzyskać na wiele sposobów. I tak, odniesienie do statku kosmicznego moglibyśmy uzyskać w naszym skrypcie także w nieco inny sposób. Deklarując w skrypcie **GameObject gracz**, nie musielibyśmy używać **[SerializeField]**, a wartość dla **gracz** mogłaby być nadana w funkcji **Start** poprzez polecenie **GameObject.FindWithTag()**, gdzie moglibyśmy użyć nadanego wcześniej temu obiektowi tagu **Player** **A**.

```
public GameObject gracz;

void Start()
{
    gracz = GameObject.FindWithTag("Player");   // A
}
```

Takie rozwiązanie dobre jest jedynie w sytuacji, kiedy mamy tylko jeden obiekt o podanym tagu. A jeśli mamy więcej obiektów o konkretnym tagu, czy nie można uzyskać odniesienia do nich z poziomu skryptu poprzez tag? Oczywiście, że można. Do przechowania takiego odniesienia do obiektu należy jednak zadeklarować tablicę, a do nadania tej tablicy wartości trzeba użyć polecenia **GameObject.FindGameObjectsWithTag** **B**. Do poszczególnych obiektów z tablicy obiektów moglibyśmy się dalej odnosić na przykład poprzez pętlę **foreach** **C**.

```
public GameObject[] gracze;
void Start()
{
    gracze = GameObject.FindGameObjectsWithTag("Player");   // B
    foreach (GameObject gracz in gracze)   // C
    {
        //instrukcje
    }
}
```

C# więcej na stronie 39

```
if (other.tag == "Player")
{
    Debug.Log("Trafiam w statek");                          H
    Instantiate(wybuch, gracz.transform.position, Quaternion.identity);
    Destroy(gracz); I
    Destroy(gameObject);
}
```

form.position, Quaternion.identity); H, gdzie **gracz.transform.position** jest odczytaniem lokalizacji gracza i to w tym miejscu pojawi się wybuch.

10 Statek kosmiczny po wybuchu nie powinien być już widoczny. Jeśli dodamy do skryptu polecenie **Destroy(gracz);** I, statek kosmiczny zniknie ze sceny. Jeśli dopiszemy też **Destroy(gameObject);**, ze sceny zniknie również asteroida.

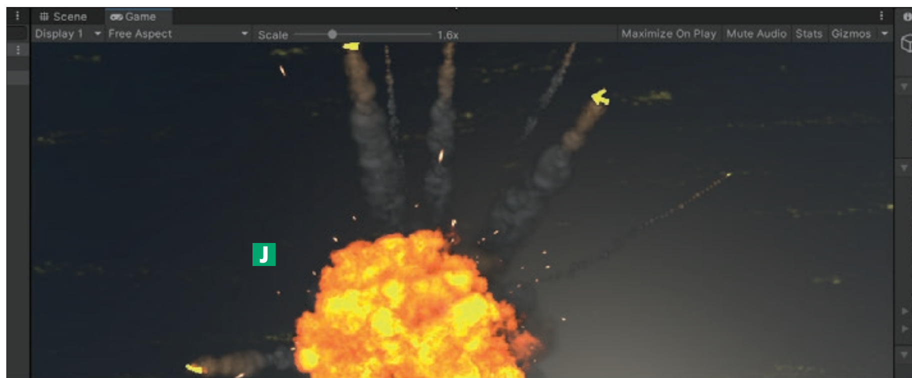
11 Przetestujmy działanie skryptu na podglądzie rozgrywki. Gdy asteroida uderzy w statek, uruchomiony zostanie efektowny wybuch J.

Więcej asteroid

Gdy w naszej grze asteroida minie statek i graczowi uda się uniknąć z nią kolizji, ta wciąż porusza się poza obszarem obejmowanym przez widok z kamery i nie jest dla nas zagrożeniem. Możemy teraz tak zmienić skrypt, by asteroida wylatująca z okna gry poprzez dolną krawędź po chwili teleportowała się do góry okna i ponownie leciała w kierunku dołu okna gry.

1 Pierwszym krokiem do realizacji tego zadania powinno być wychwycenie momentu teleportacji asteroidy. Jeśli uruchomimy podgląd gry, zobaczymy, że gdy asteroida porusza się w stronę dołu okna, zmienia się jej współrzędna **z**. To właśnie osiągnięcie przez nią konkretnej wartości powinno być momentem kluczowym dla wywołania teleportacji asteroidy. Gdy jej współrzędna **z** jest mniejsza niż -5, asteroidy nie widać już w obszarze obejmowanym przez kamerę. Dodajmy zatem instrukcję warunkową sprawdzającą, czy współrzędna z asteroidy jest mniejsza niż -5. Wartość współrzędnej **z** obiektu, do którego załączony jest skrypt, odczytamy dzięki **transform.position.z** A. Takie sprawdzanie powinno odbywać się po każdym przesunięciu się asteroidy, zatem instrukcja warunkowa powinna być zapisana w funkcji **Update**.

2 By przenieść asteroidę do góry okna gry, czyli ponad obszar obejmowany przez widok z kamery, należy nadać jej nową pozycję. Nowa pozycja to nowy trójwymiarowy wektor, który utworzy-



```
void Update()
{
    transform.position = transform.position + Vector3.back * Time.deltaTime;
    if (transform.position.z <=-5)
    {                    A
        |
    }
```

my, zapisując **transform.position = new Vector3();**.

```
if (transform.position.z <=-5)
{
    transform.position = new Vector3();
}
```

3 Tworząc ten nowy wektor, powinniśmy podać trzy wartości, z których każda będzie odpowiadać nowym współrzędnym asteroidy – odpowiednio na osiach **X**, **Y** i **Z**. By utrudnić zadanie graczowi, możemy nadać asteroidzie taką współrzędną **x**, by leciała ona prosto na obiekt sterowany przez gracza. Jej współrzędna **x** powinna być wtedy taka sama jak współrzędna **x** statku kosmicznego, którą odczytamy z wartości **gracz.transform.position.x** i taka właśnie wartość B powinna pojawić się jako pierwszy element nowego **Vector3**.

4 Druga ze współrzędnych nowej lokalizacji asteroidy to **y**, czyli wysokość (jak blisko kamery), na jakiej leci asteroida. Ona tak naprawdę nie musi się zmieniać, zatem w nowym wektorze możemy uzupełnić ją jako **transform.position.y** C, czyli wartość dotychczasowej współrzędnej **y**.

5 Ostatnia ze współrzędnych, czyli **z**, mówi o tym, jak wysoko w oknie gry ma pojawić się asteroida. Wartość **6** D sprawi, że asteroida pokaże się już ponad górną krawędzią okna gry.

6 Zapiszmy zmiany w skrypcie i przetestujmy je na podglądzie rozgrywki. Zobaczymy, że po pokonaniu asteroidy pojawia się ona ponownie u góry okna gry.

```
if (transform.position.z <=-5)
{                                                         B
    transform.position = new Vector3(gracz.transform.position.x, , );
}
```

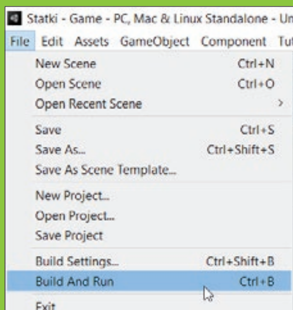
```
if (transform.position.z <=-5)
{                                                                          C          D
    transform.position = new Vector3(gracz.transform.position.x, transform.position.y, 6);
}
```

GOTOWY PLIK Z GRĄ

Tak przygotowaną grę można byłoby już wyeksportować jako efekt naszej pracy.

1 Jeśli chcemy uzyskać gotowy plik z grą, z menu górnego, z opcji **File**, wybieramy **Build And Run**.

2 Zostanie otwarte okno dialogowe pozwalające na określenie lokalizacji, w jakiej chcemy zapisać gotową grę.



3 Po wybraniu folderu do zapisania gry utworzony zostanie plik wykonywalny z grą, która zostanie od razu uruchomiona i będziemy mogli ją przetestować.



Na DVD znajdziemy plik projektu przedstawionego w tym rozdziale

6 Gra z widokiem pierwszoosobowym

Kolejny rozdział to kolejny projekt w Unity. W poprzednim projekcie stworzyliśmy grę z widokiem z góry. Tym razem nasza gra będzie miała widok pierwszoosobowy

Bazą projektu ponownie może być scena z menu. Powtarzamy zatem kroki opisane w rozdziale o tworzeniu menu z przyciskiem startującym grę i tak jak w poprzednim rozdziale dodajemy osobną scenę z rozgrywką, by uruchomiła się ona po kliknięciu na przycisk Start. W dalszej części tego rozdziału skupimy się już na budowie sceny z rozgrywką.

Gra z widokiem pierwszoosobowym

W tym rozdziale opiszemy, jak stworzyć grę z widokiem pierwszoosobowym.
To specyficzny rodzaj rozgrywki, w którym kamera pełni funkcję oczu gracza. W naszej rozgrywce gracz będzie poruszał się po lesie, w którym ukryto bombę.

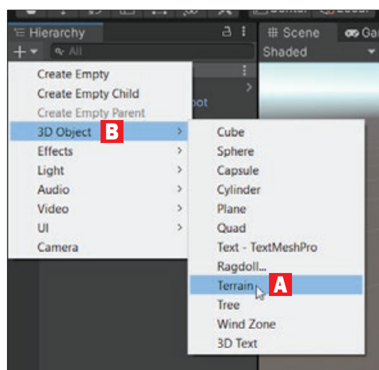


Szukamy bomby ukrytej w lesie – gra tworzona w tym rozdziale

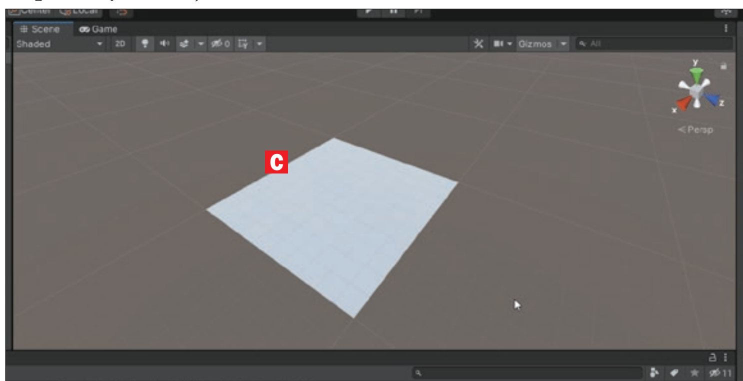
Tworzymy mapę gry

By móc poruszać postacią po mapie, niezbędny będzie kontroler. Możemy go pobrać z Asset Store. Jednak zanim to zrobimy, powinniśmy zająć się przygotowaniem podstawy mapy, po której będziemy się poruszać.

1 Unity wśród obiektów, jakie możemy umieszczać na scenie, oferuje **Terrain** (czyli teren). Wykorzystamy ten obiekt. Dodamy go do naszej sceny. Klikamy na plus w rogu sekcji **Hierarchy**, aby rozwinąć kategorie obiektów do wyboru. Poszukiwany przez nas obiekt **Terrain** **A** jest obiektem trójwymiarowym, więc znajduje się w grupie **3D Object** **B**.



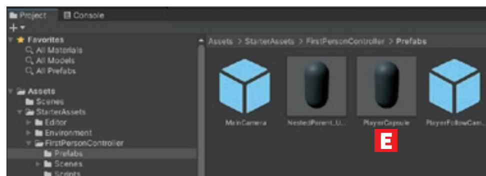
2 To, co zostało dodane do sceny, z pozoru wygląda jak zwykła płaszczyzna **C**, jednak obiekt **Terrain** daje nam znacznie większe możliwości niż „prawdziwa" płaszczyzna – **Plane**. Tylko w początkowej fazie obiekt ten jest płaski. Dzięki opcjom ukrytym w panelu jego właściwości możemy zmienić ukształtowanie terenu czy też wypełnić go drzewami. Tym jednak zajmiemy się w dalszej części realizacji tego projektu. Najpierw – kontroler postaci.

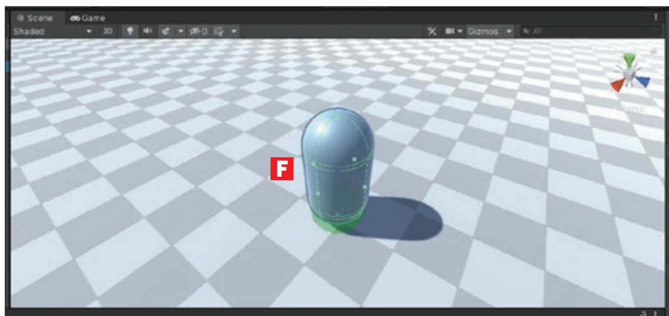


3 Przechodzimy do strony internetowej z Asset Store i znajdujemy zasób **Starter Assets – First Person Character Controller** **D**, pobieramy go i importujemy do projektu.

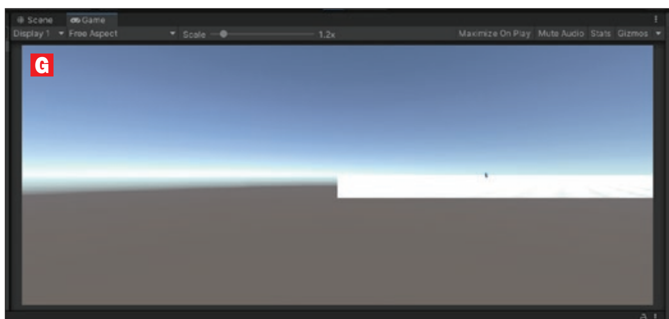


4 Paczka zasobów znajduje się w folderze **StarterAssets**, a interesujący nas kontroler postaci znajdziemy, wchodząc do folderu **FirstPersonController**, a następnie **Prefabs**. By skorzystać z gotowego kontrolera postaci, przeciągamy na scenę prefabrykat **PlayerCapsule** **E**.





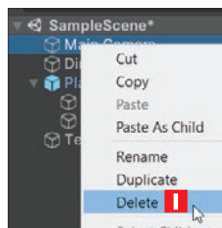
5 Obiekt w kształcie kapsuły **F** będzie pełnił rolę obiektu – gracza. Nie przejmujmy się tym, że nie wygląda on zachwycająco. Ponieważ gra jest z widokiem pierwszoosobowym – nie będziemy widzieć tego obiektu.



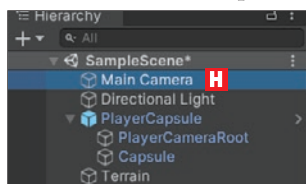
6 Umieszczony na scenie obiekt zawiera kontroler postaci – to oznacza, że obiektem takim można sterować. Kontroler ma domyślne przypisane poruszanie się po terenie zarówno za pomocą strzałek, jak i za pomocą klawiszy W A S D. Postać potrafi nawet skakać (skok wywołamy, wciskając spację). Można byłoby już teraz sprawdzić, czy faktycznie jesteśmy w stanie sterować obiektem po uruchomieniu podglądu – jednak to, co w tym momencie moglibyśmy zobaczyć na podglądzie **G**, będzie mocno odbiegać od naszych oczekiwań. Na podglądzie zobaczymy bowiem widok z boku na nasz teren, na którym może być widać niewielką kapsułę, czyli obiekt gracza. I choć można nim sterować i przesuwać go po terenie – to całość miała wyglądać zupełnie inaczej.

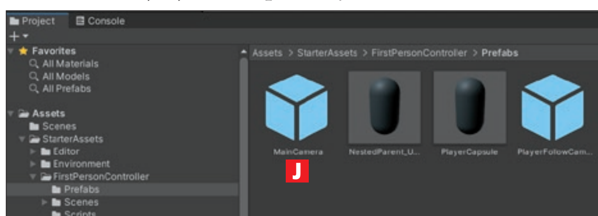
7 Niezgodny z oczekiwaniami sposób wyświetlania rozgrywki to wynik obecności na scenie domyślnej kamery, czyli obiektu **Main Camera** **H**. Obiekt ten nie porusza się i nie obraca wraz z kapsułą. Domyślną kamerę powinniśmy zastąpić inną – połączoną z obiektem – graczem.

8 Przed dodaniem nowej kamery do sceny trzeba pozbyć się obecnej. Rozwijamy menu kontekstowe, klikając prawym przyciskiem myszy na obiekt kamery w panelu **Hierarchy**. Z dostępnych opcji wybieramy **Delete** **I**, co doprowadzi do usunięcia kamery ze sceny.

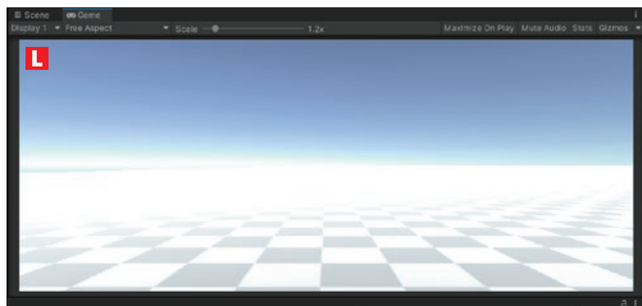


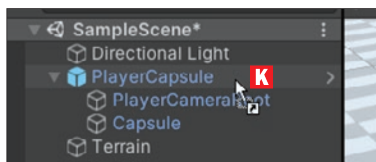
9 W zasobach projektu, w folderze **Prefabs**, z którego braliśmy prefabrykat obiektu PlayerCapsule, znajduje się też prefabrykat **MainCamera** **J**.





Wykorzystamy go jako kamerę, która będzie pełniła rolę oczu gracza. By tak było, powinna być ona dodana do sceny w taki sposób, by była jednym z elementów budujących naszą kapsułę – przeciągamy prefabrykat kamery na **PlayerCapsule** K na liście obiektów sceny.



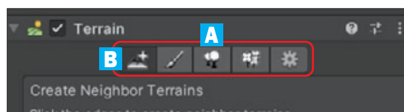


10 Gdy teraz uruchomimy podgląd rozgrywki, zobaczymy, że widok z kamery L zmienia się wraz z ruchem kursora myszy na boki. Dodatkowo zmienia się on wraz z ruchem kapsuły, czyli po wciskaniu na klawiaturze strzałek czy spacji podczas skoku.

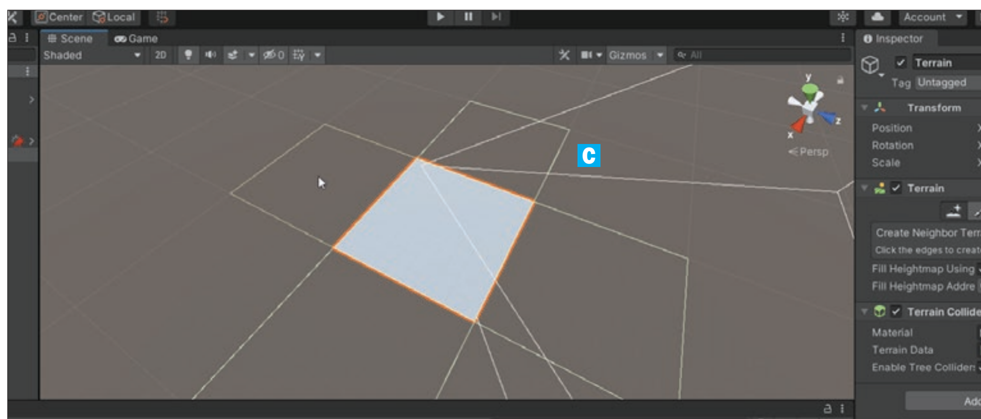
Modelowanie mapy gry

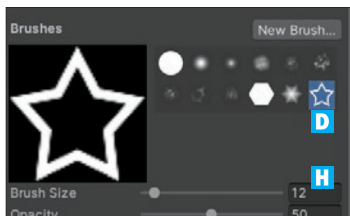
Skoro możemy poruszać się już po mapie, to w tej części rozdziału zajmiemy się odpowiednim przerobieniem obiektu **Terrain** tak, by stał się on pełnoprawną mapą gry. Edycję terenu przeprowadzimy z poziomu jego właściwości. W nich znajdziemy zestaw przycisków A.

1 Wybranie pierwszego z przycisków B pozwala na powiększanie terenu. Na scenie zaznaczone będą kwadraty C odpowiadające kolejnym częściom terenu, o jakie można powiększyć obiekt **Terrain**. W miarę powiększania terenu stają się aktywne kolejne kwadraty, o które można go dalej powiększać.

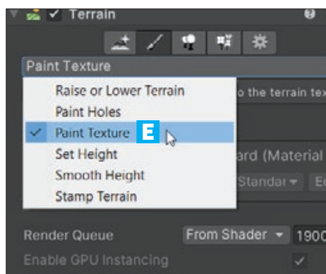


2 Gdy wybierzemy drugi z przycisków, we właściwościach obiektu **Terrain** pojawi się zestaw zmienionych opcji. Wśród nich znajduje się możliwość wyboru kształtu pędzla D – tym pędzlem można „malować te-

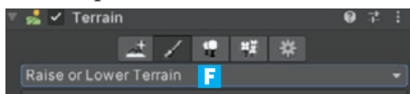




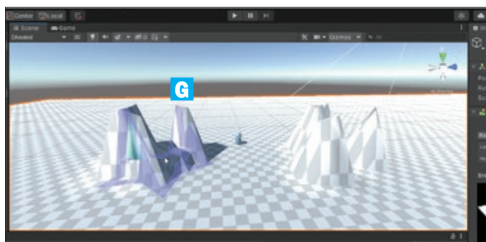
ren”. Co to oznacza? Otóż mamy kilka możliwości edycji terenu, które możemy wybierać z poziomu rozwijanej listy **E** pod zestawem przycisków we właściwościach.



3 Jeśli z listy wybierzemy opcję **Raise or Lower Terrain** **F**, będziemy mogli regulować wysokość terenu. Wystarczy kliknąć, przytrzymać lewy przycisk myszy i przesuwać pędzlem po terenie na scenie. Teren będzie się podnosił **G**.



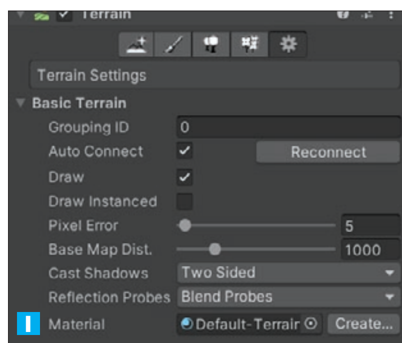
4 Rozmiar pędzla można regulować poprzez suwak **Brush Size** **H** pod zestawem pędzli do wyboru.



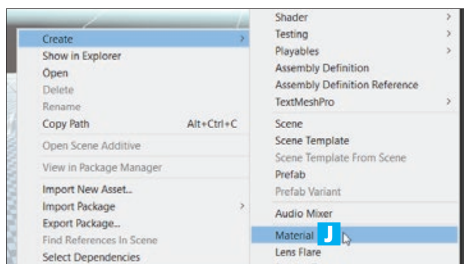
5 By obniżyć teren, postępujemy tak samo, jak podczas jego podnoszenia, ale przytrzymujemy dodatkowo wciśnięty klawisz shift. Teren nie obniży się jednak poniżej poziomu bazowego, od którego rozpoczynaliśmy jego podnoszenie.

6 Trzeci z przycisków we właściwościach terenu pozwala na skorzystanie z opcji masowego zadrzewiania terenu. To bardzo przydatne narzędzie do budowania lasu. Jednak by zbudować las, niezbędne będzie posiadanie trójwymiarowych modeli drzew – drzewami zajmiemy się w dalszej części tego rozdziału.

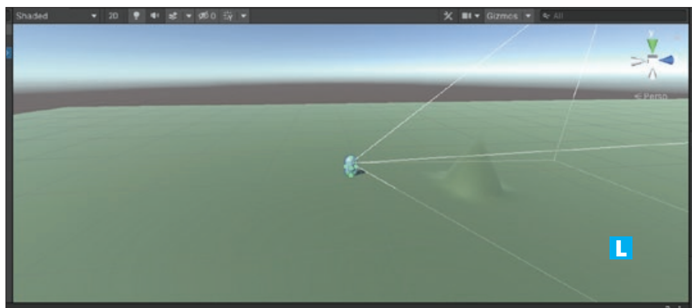
7 Ostatni z przycisków we właściwościach terenu – przycisk z kołem zębatym – to zbiór ogólnych ustawień terenu. Wśród nich znajduje się opcja **Material** **I**, która decyduje o tym, jakim materiałem pokryta jest powierzchnia terenu.

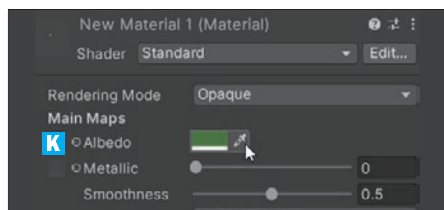


8 Jednak by takim materiałem pokryć teren, musimy go mieć. Materiał można stworzyć samodzielnie. W zasobach projektu rozwijamy menu kontekstowe i z opcji **Create** wybieramy pozycję **Material** **J**.



9 We właściwościach nowego materiału możemy zmienić opcję **Albedo** **K**, która ma główny wpływ na barwę materiału. Kolor dla tej właściwości możemy wybrać z palety, można też pobrać go z pomocą opcji pipety – na podstawie dowolnego koloru wskazanego w obrębie ekranu.





10 Gdy materiał jest już odpowiednio zmodyfikowany, możemy połączyć go z odpowiednią opcją z właściwości terenu, przeciągając materiał z zasobów projektu do pola **Material** **I**.

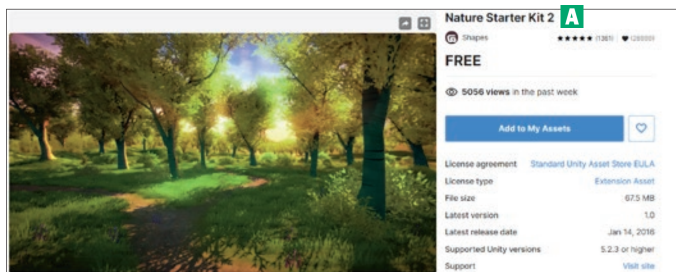
11 Po połączeniu materiału z opcją cały teren pokryty będzie już naszym materiałem **L**.

Masowe zadrzewianie

Teraz możemy zająć się zadrzewianiem naszego terenu, by stworzyć las. Pierwszym krokiem zmierzającym do dodania drzew do sceny powinno być umieszczenie ich w zasobach projektu. Drzewa pobierzemy z Asset Store.

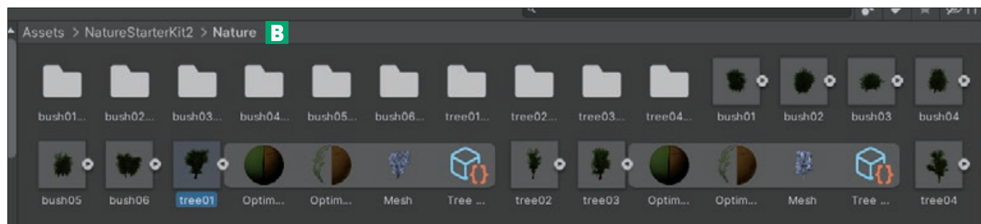
1 Znajdujemy paczkę zasobów o nazwie **Nature Starter Kit 2** **A**. Pobieramy ją i importujemy do projektu.

2 Po dodaniu paczki zasobów do projektu znajdujące się w niej zasoby trafiają do

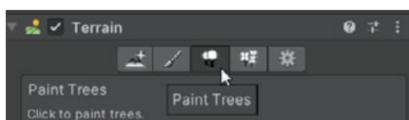


folderu **NatureStarterKit2**. Modele drzew są natomiast w znajdującym się tam folderze **Nature** **B**.

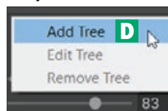
3 By móc masowo, jednym kliknięciem wstawić wiele drzew na określonym

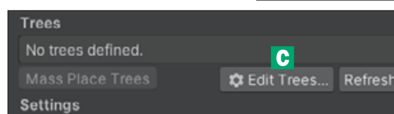


przez nas obszarze, powinniśmy połączyć wybrany model drzewa z opcją masowego zadrzewiania. We właściwościach terenu wybieramy opcję **Paint Trees**.

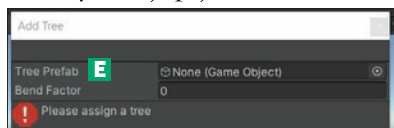


4 W sekcji **Trees** widzimy informację **No trees defined**, która mówi o tym, że nie mamy jeszcze zdefiniowanych drzew do masowego zadrzewiania. By takie drzewo wybrać, klikamy na **Edit Trees** **C**, a następnie na **Add Tree** **D**.

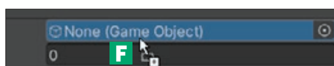




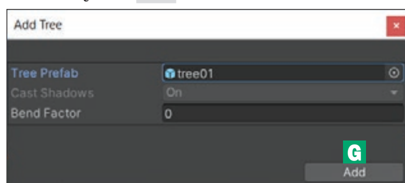
5 Otwarte zostanie nowe okno. Na prefabrykat drzewa wskazuje tam pole **Tree Prefab** **E**, które jest puste. Można uzupełnić je na dwa sposoby. Pierwszy z nich to kliknięcie na kółko obok opcji, co spowoduje otwarcie okna z zestawieniem wszystkich prefabrykatów z zasobów projektu, których można użyć w tej opcji.



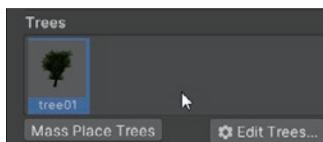
6 Druga z możliwości to przeciągnięcie do pola **F** wybranego prefabrykatu drzewa bezpośrednio z zasobów projektu.



7 Gdy prefabrykat jest już wybrany, należy kliknąć na **Add** **G**.



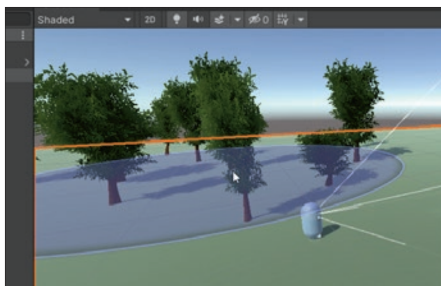
8 Drzewo zostanie wyświetlone w sekcji **Trees** ustawień terenu, jednak nie jest ono jeszcze dodane do sceny.



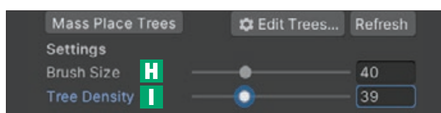
9 Drzewa mogą pojawić się na scenie na obszarze wyznaczonym przez pędzel (koło).



10 Gdy klikniemy na scenę, drzewa zostaną wstawione.

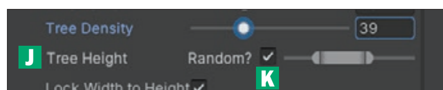


11 Mamy wpływ na to, jak duży jest obszar zajmowany przez pędzel – dzięki suwakowi **Brush Size** **H**.



12 We właściwościach zadrzewiania mamy też inny suwak – **Tree Density** **I** – za jego pomocą możemy kontrolować to, jaka ma być gęstość zadrzewienia na wyznaczonym przez pędzel obszarze.

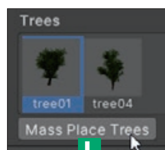
13 Również znajdująca się we właściwościach opcja **Tree Height** **J** pozwala



na określenie, czy wysokość drzew ma być losowa (**Random** K). Obok pola pozwalającego określić, czy wykonywać losowanie wysokości, mamy podwójny suwak zakresu umożliwiający określenie minimalnej i maksymalnej wysokości drzewa. Ten mechanizm jest bardzo przydatny. Dzięki różnym wysokościom drzew uzyskany przez nas efekt będzie przypominał prawdziwy las, w którym każde drzewo wygląda inaczej.

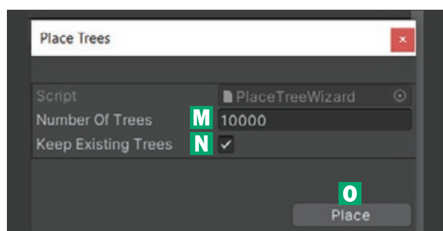
14 Pędzel maluje zawsze drzewa jednego rodzaju. Jeśli powtórzymy operację dodawania prefabrykatu drzewa, kolejne drzewo ukaże nam się we właściwościach terenu. Pędzlem umieszczamy tylko drzewa wybranego rodzaju.

15 Nieco inaczej działa opcja masowego zadrzewiania dostępna pod przyciskiem **Mass Place Trees** L.

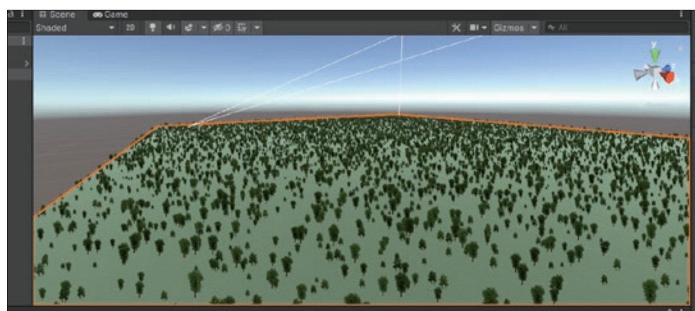


16 Po jej wyborze w nowym oknie możemy określić, jak dużo drzew chcemy wstawić na całym terenie (liczbę wpisujemy w pole **Number of Trees** M).

17 W oknie tym dostępna jest też opcja **Keep Existing Trees** N. Jeśli jest zaznaczona, istniejące wcześniej na terenie drzewa nie zostaną usunięte. Gdy usuniemy jej zaznaczenie, „starsze” drzewa zostaną wykasowane przy przeprowadzeniu masowego zadrzewiania.



18 By masowe zadrzewianie zostało wykonane, klikamy na **Place** O.

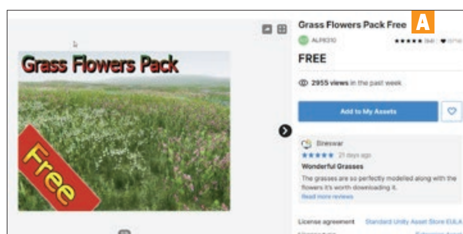


19 Po chwili cały teren będzie pokryty drzewami. W opcji masowego zadrzewiania wykorzystywane są wszystkie modele drzew dodane wcześniej do sekcji **Trees**, dzięki czemu możemy otrzymać bardziej różnorodny las.

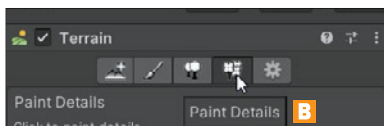
Malujemy trawę i kwiaty

Nasz las przypomina jeszcze „pustynię z drzewami” – jest pokryty gładkim materiałem, a oprócz drzew nie rosną w nim żadne inne rośliny. Możemy to zmienić. Edytor terenu w Unity ma opcję dodawania traw i kwiatów. By jednak móc wstawiać tego typu obiekty, wcześniej trzeba je dodać do zasobów projektu.

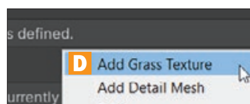
1 Znajdujemy w Asset Store paczkę zasobów **Grass Flowers Pack Free** A, pobieramy ją i importujemy do projektu.

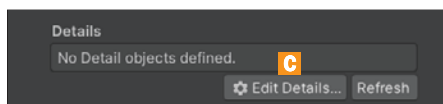


2 We właściwościach obiektu **Terrain** należy kliknąć na przycisk **Paint Details** **B**, aby przejść do opcji wstawiania kwiatów i traw. Edytor terenu dla traw i kwiatów ma inne opcje wstawiania niż te, jakie poznaliśmy w wypadku drzew, między innymi dlatego, że na te obiekty może mieć wpływ wiatr – to znaczy mogą się one kołysać na wietrze.

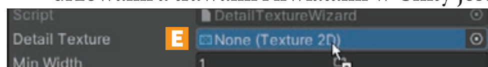


3 W sekcji **Details** we właściwościach nie ma jeszcze dodanych żadnych obiektów. Aby je dodać, klikamy na przycisk **Edit Details** **C**, a potem na **Add Grass Texture** **D**.





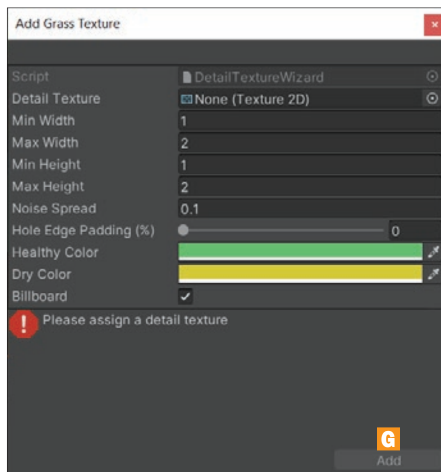
4 Spowoduje to otwarcie nowego okna dodawania tekstur. Kolejną różnicą między drzewami a trawami i kwiatami w Unity jest



to, że drzewa są obiektami trójwymiarowymi, natomiast obiekty, które teraz będziemy dodawać, są dwuwymiarowymi teksturami, które silnik Unity przetwarza w taki sposób, by sprawiały wrażenie trójwymiarowych. Teksturę trawy lub kwiatów należy przeciągnąć z zasobów projektu do pola **Detail Texture** **E**.

5 Pobrana paczka zasobów jest widoczna w zasobach projektów jako folder **GrassFlowers**, w którym znajduje się katalog **Textures**, a w nim z kolei kolejny folder o nazwie **GrassFlowers**. To w nim znajdują się tekstury **F**, z których możemy wybrać tę, którą chcemy umieścić na scenie.

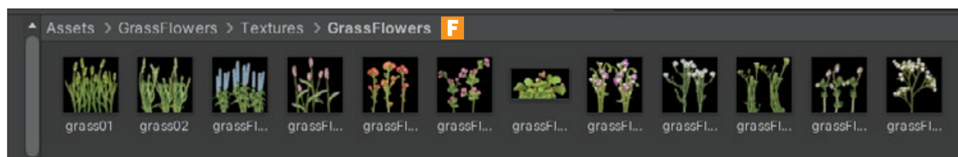
6 By zamknąć okno dodawania, klikamy na przycisk **Add** **G** widoczny w rogu okna.



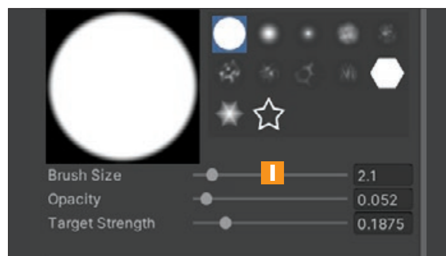
7 Wybrane przez nas kwiaty bądź trawę umieszczamy na terenie, malując je pędzlem **H**.

8 Podobnie jak w przypadku drzew mamy we właściwościach zestaw suwaków, po-

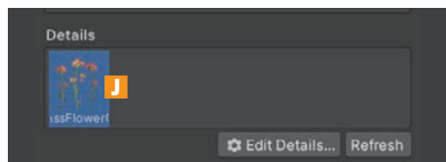




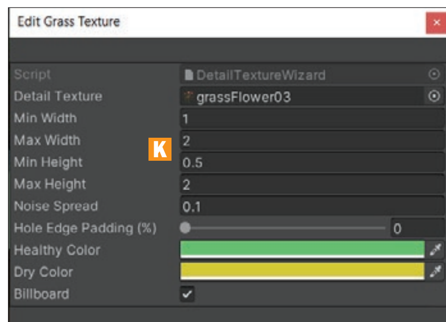
przez które możemy wpłynąć na parametry wstawiania, na przykład możemy zmienić wielkość pędzla, korzystając z suwaka **Brush Size** **I**.



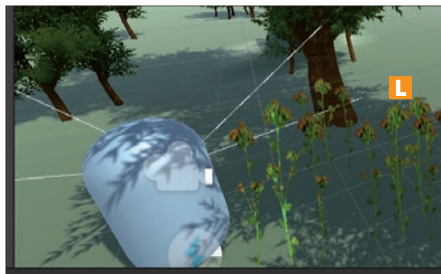
9 Bezpośrednio we właściwościach nie widać jednak opcji, która pozwalałaby na zmianę wysokości traw i kwiatów. Na szczęście istnieje sposób, aby ją zmienić, nawet wtedy, gdy trawy i kwiaty są już dodane do terenu. Klikamy dwukrotnie na teksturę **J** w panelu **Details**.



10 Otworzymy w ten sposób okno edycji dodanego zasobu, które jest bardzo podobne do okna, w którym dodawaliśmy zasób do terenu. Zarówno w jednym, jak i w drugim oknie zobaczymy pola podpisane jako **Min Height** i **Max Height** **K**, które odpowiadają za minimalną i maksymalną wysokość trawy bądź kwiatów. Zmieniając



wpisane tam liczby, możemy od razu zobaczyć, jak na terenie zmieniają się wysokości edytowanych roślin. Na przedstawionym przykładzie **L** widać, jak powiększyły się rośliny, gdy pole **Max Height** dostało wartość **4**, zamiast znajdującej się w nim wcześniej wartości 2.



11 Wiemy już, że możemy wpływać na to, jak na dodaną przez nas roślinność oddziałuje wiatr. Grupa właściwości dotyczących tego zachowania będzie widoczna dopiero po kliknięciu we właściwościach terenu na ostatni przycisk z zestawu przycisków – z kołem zębatym. Ustawienia dotyczące działania wiatru znajdują się w grupie **Wind Settings for Grass**.

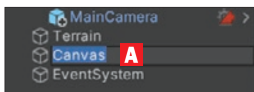


Dla lepszej obserwacji tego, jak zmiana wartości na suwakach wpływa na zachowanie roślin, możemy uruchomić podgląd rozgrywki i nakierować obiekt gracza w taki sposób, by kamera obejmowała rosnące na terenie rośliny. Wtedy będziemy wszystko obserwować na bieżąco.
Należy jednak przy tym pamiętać, że jakiekolwiek zmiany we właściwościach, wprowadzane w trybie testowania gry, po zatrzymaniu testowania wrócą do stanu sprzed testowania – dlatego jeśli chcemy trwale zmienić zachowanie roślin na wietrze, zapamiętajmy ustawienie suwaków, które najbardziej nam odpowiada, i ustawmy suwaki zgodnie z nim po zakończeniu testowania.

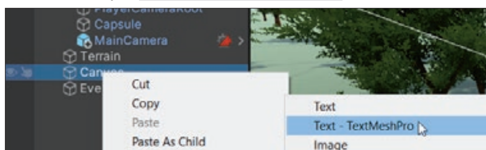
Odliczanie czasu

Nasza gra docelowo ma polegać na odnalezieniu bomby, nim ta wybuchnie. Zadanie musimy zatem wykonać w ściśle określonym czasie. W tej części rozdziału zajmiemy się stworzeniem mechanizmu odliczającego i wyświetlającego w oknie gry czas, jaki pozostał do wybuchu bomby.

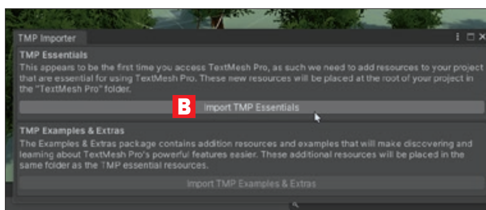
1 Pierwszym krokiem będzie dodanie do sceny obiektu **Canvas** **A**, czyli płaszczyzny, na której możemy umieszczać elementy UI.



2 Do wyświetlenia na tej płaszczyźnie tekstu posłuży nam kolejny obiekt z grupy UI, czyli **Text – TextMeshPro**.

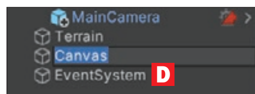


3 Po wyborze obiektu do dodania program wyświetli nam kolejne okno, które pozwala zaimportować do projektu paczkę zasobów obsługujących obiekt tekstowy, klikając na **Import TMP Essentials** **B**.

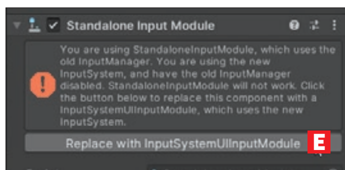


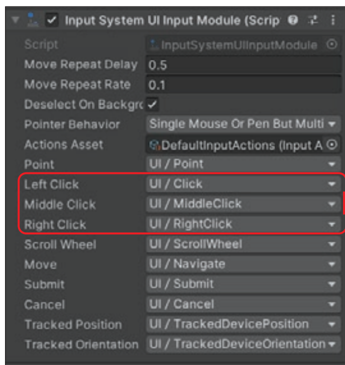
4 Po jego dodaniu niemożliwe będzie uruchomienie podglądu rozgrywki – pojawi się komunikat o błędzie **C**.

5 Pozbycie się tego błędu jest możliwe z poziomu właściwości obiektu **EventSystem** **D**, który został automatycznie dodany do sceny wraz z obiektem **Canvas**.

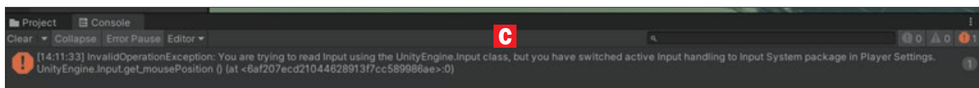


6 Znajdujemy sekcję **Standalone Input Module** i wybieramy w niej przycisk **Replace with InputSystemUIInputModule** **E**, co zmienia nieco właściwości obiektu, wprowadzając do nich pola **F** pozwalające na mapowanie instrukcji wejścia.

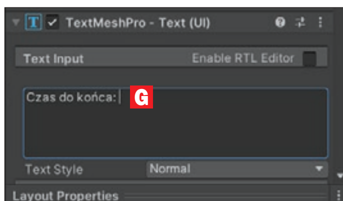




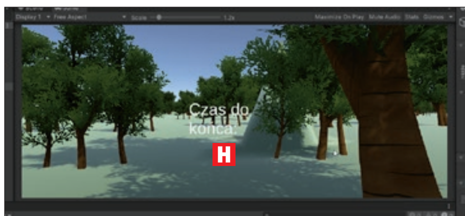
7 We właściwościach obiektu tekstowego możemy edytować pole tekstowe, w którym można wpisać frazę: **Czas do końca:** **G** będącą początkiem tekstu, jaki będzie wyświetlany na tym obiekcie. Po niej powinniśmy dopisać, ile sekund pozostało do wybuchu bomby. Wartość liczbowa zostanie



tam dopisana z poziomu skryptu odliczającego czas.

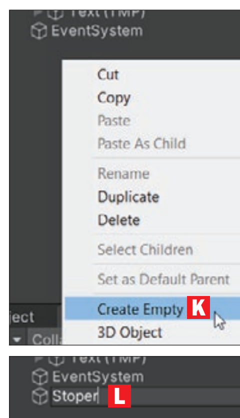


8 Gdy dodajemy **Canvas** do sceny, jest on umieszczany w taki sposób, że znajdujące się na nim obiekty zawsze są widoczne na ekranie podczas rozgrywki. Taki obiekt bezpośrednio po dodaniu jest widoczny na środku okna rozgrywki **H**.



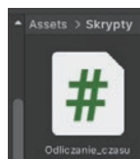
9 We właściwościach obiektu tekstowego możemy zmienić jego lokalizację poprzez zmianę wartości pól **PozX** i **PozY** **I**. Podobny efekt możemy uzyskać, przeciągając obiekt na scenie z wykorzystaniem wyświetlanych na nim kolorowych strzałek **J**.

10 Do obsługi odliczania czasu wykorzystamy oddzielny obiekt na scenie. Unity pozwala nam na tworzenie obiektów, które są niewidoczne, ale niosą za sobą pewne zachowania i wartości. Takim obiektem będzie stoper odliczający czas do wybuchu bomby. Tworzymy nowy obiekt na scenie, wybierając z menu kontekstowego na panelu **Hierarchy** opcję **Create Empty** **K**.



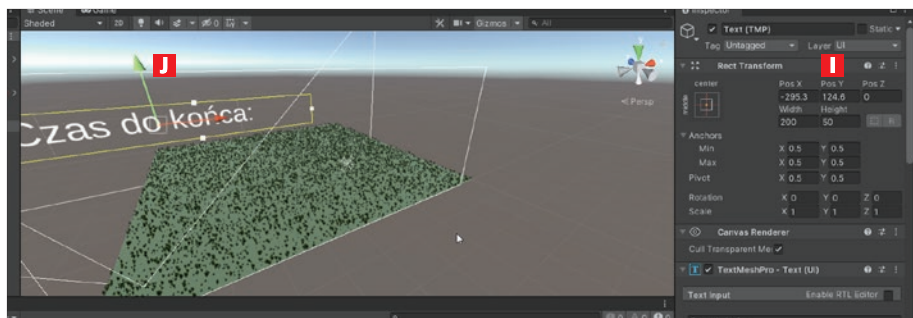
11 Nowemu obiektowi nadajemy nazwę **Stoper** **L**.

12 W zasobach projektu zakładamy nowy folder o nazwie **Skrypty**. W nowym folderze zapisujemy skrypt i nadajemy mu nazwę **Odliczanie_czasu**.



13 Otwieramy utworzony skrypt. Trzeba w nim zapisać, jaka będzie początkowa wartość czasu, który ma upłynąć do wybuchu bomby. Czas ten powinien być przechowywany przez skrypt w polu o nazwie **czas**. Wartość czasu może być wyrażona liczbą całkowitą i reprezentować liczbę sekund. Zatem typem danych użytym do jej przechowania może być **int**. Możemy przyjąć, że na początku gracz dostanie **60** **M** sekund na wykonanie zadania. Jeśli chcemy móc łatwo zmieniać tę liczbę bez konieczno-

```
public class Odliczanie_czasu : MonoBehaviour
{
    [SerializeField] public int czas = 60; M
```



ści wchodzenia w edycję skryptu, możemy oznaczyć to pole jako **[SerializeField]**.

14 Podobnie możemy dodać do skryptu obiekt typu **TextMeshProUGUI** **N**, który będzie pełnił rolę łącznika pomiędzy skryptem stopera a obiektem graficznym do wyświetlania czasu pozostałego do wybuchu.

```
[SerializeField] public int czas = 60;
[SerializeField] public TextMeshProUGUI t; N
// Start is called before the first frame upd
```

15 By móc użyć w skrypcie obiektu tego typu, należy dodać do skryptu przestrzeń nazw **TMPro** **O**.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using TMPro; O
```

16 W funkcji **Start** powinniśmy zapisać instrukcję, która będzie mogła zmienić tekst na obiekcie tekstowym. Jeśli obiekt nazywa się **t**, to napis na nim ustawimy, nadając wartość dla **t.text**. Tą wartością powinien być początkowy napis na obiekcie, czyli **Czas do końca:**, do którego dodamy wartość zapisanego w skrypcie czasu. Pamiętajmy jednak, że czas jest zmienną liczbową, a napis – łańcuchem znaków. Nie możemy przeprowadzić dodawania danych dwóch różnych typów. Zatem liczbę reprezentującą czas możemy przerobić na tekst poprzez dopisanie do niej **.ToString()** **P**.

```
void Start()
{                                              P
    t.text = "Czas do końca: " + czas.ToString();
}
```

17 W obrębie skryptu zadeklarujmy jeszcze **private float odliczony = 0f;** **Q** – będzie to prywatne pole (czyli takie, do którego z poziomu innych obiektów nie będziemy mogli się dostać). Dodatkowo, jeśli choć raz użyliśmy w skrypcie **[SerializeField]**, to wszystkie pola oznaczone jako **public** będą wyświetlane w oknie edytora i można będzie nadawać im nowe wartości. Nie pokażą się tam za to pola oznaczone jako **private**.

```
// Start is called before the first frame update

private float odliczony = 0f; Q
void Start()
```


18 W polu **odliczony** będzie przechowywany czas, jaki upłynął. Gdy zostanie odliczone upłynięcie jednej sekundy, zmieni się napis na obiekcie tekstowym.

19 Zapisywanie odliczonego czasu powinno odbywać się wraz z wyświetleniem każdej klatki rozgrywki, a co za tym idzie, powinniśmy zapisać je w funkcji **Update**. Tam **odliczony** powinien dostawać nową wartość, która będzie sumą poprzedniej jego wartości i czasu, jaki upłynął od wyświetlenia ostatniej klatki, zapisanego jako **Time.deltaTime** **R**.

```
// Update is called once per frame
void Update()
{                                           R
    odliczony = odliczony + Time.deltaTime;
```

20 Następnie powinniśmy utworzyć instrukcję warunkową, poprzez którą sprawdzimy, czy odliczony czas jest większy od jednej sekundy lub jej równy **S**.

```
odliczony = odliczony + Time.deltaTime;
if (odliczony >= 1f) S
{
    |
}
```

21 Gdy warunek w instrukcji jest spełniony, wartość pola **czas** powinna zmniejszyć się o **1** **T**, a odliczony przez nas czas powinien zostać wyzerowany, dzięki czemu ponowne spełnienie warunku nastąpi dopiero po upłynięciu kolejnej sekundy, a nie „zawsze, po upłynięciu sekundy”.

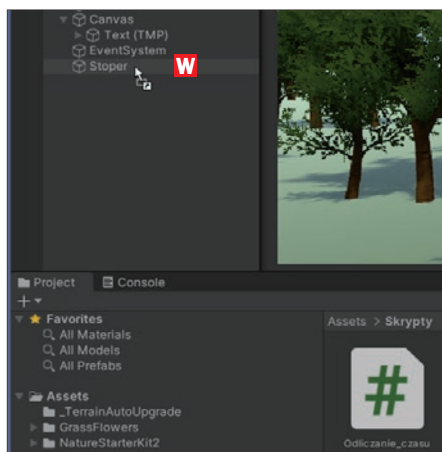
```
if (odliczony >= 1f)
{
    czas = czas - 1; T
    odliczony = 0f;
}
```

22 Po zakończeniu instrukcji warunkowej, ale wciąż w obrębie funkcji

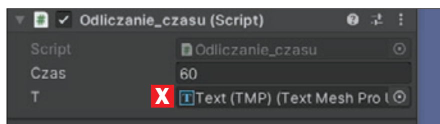
Update, powinniśmy zmodyfikować napis wyświetlany na obiekcie tekstowym. Ponownie należałoby tu umieścić fragment tekstu połączony z liczbą, czyli wykonać identyczną instrukcję zmieniającą **t.text** U, jaką wykonaliśmy w funkcji **Start**.

```
        odliczony = 0f;
    }
U   t.text = "Czas do końca: " + czas.ToString();
}
```

23 By skrypt zaczął działać, konieczne jest podłączenie go do obiektu na scenie. Obiekt ten to **Stoper**. Przeciągamy skrypt W z zasobów projektu na obiekt **Stoper** w panelu **Hierarchy**.



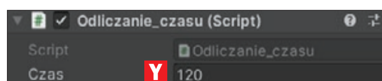
24 W panelu właściwości obiektu **Stoper** uzupełniamy jeszcze pole **T**, przeciągając do niego obiekt tekstowy z panelu **Hierarchy** X.

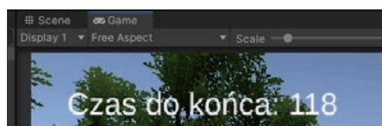


25 Gdy uruchomimy podgląd gry, zobaczymy już, że czas na ekranie odlicza się co sekundę w dół.



26 We właściwościach stopera znajduje się jeszcze pole **Czas** – zmieniając domyślną w nim wartość 60, możemy zmienić w dowolnym momencie czas Y pozostały do wybuchu bomby. Jeśli przetestujemy rozgrywkę po zmianie wartości w polu, zobaczymy, że podany przez nas czas będzie teraz odliczany w oknie gry.



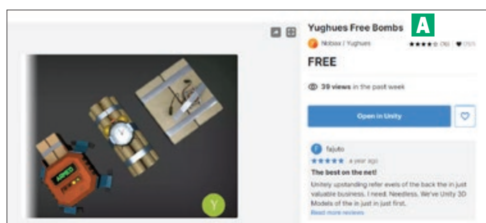


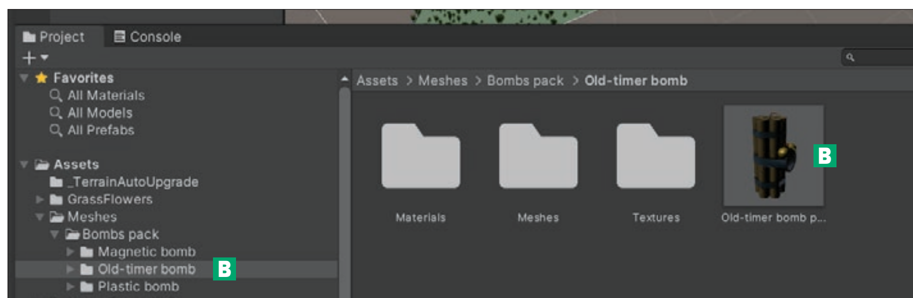
Tworzymy bombę

Umieszczenie bomby na scenie

Aby dokończyć naszą grę, potrzebujemy jeszcze na scenie bomby.

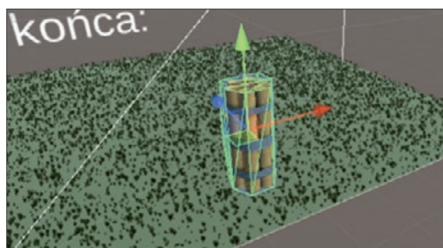
1 Możemy tu także skorzystać z zasobów Asset Store i pobrać ze sklepu z zasobami paczkę **Yughues Free Bombs** A.





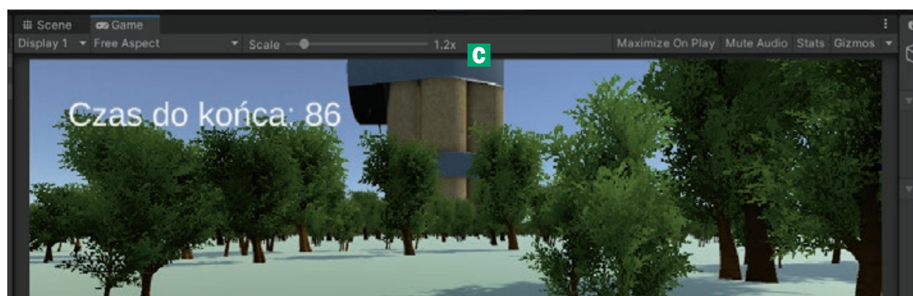
2 Zasoby z tej paczki są widoczne w zasobach projektu w folderze **Meshes**. Są tam trzy katalogi z bombami. W naszym przykładzie wykorzystamy **Old-timer bomb** **B**.

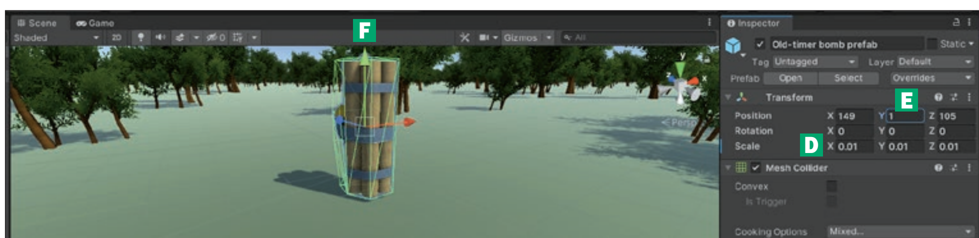
3 Przeciągamy z zasobów na scenę wybraną bombę.



4 Jeśli przeciągniemy bombę na scenę, zobaczymy, jak dużym jest ona obiektem – dobrze widać to szczególnie po uruchomieniu podglądu rozgrywki, gdzie bomba wygląda jak olbrzymi budynek w oddali **C**.

5 By bomba miała rozmiar bardziej pasujący do reszty obiektów na scenie, należy ją zmniejszyć. W tym celu edytujemy pola z części **Scale** we właściwościach bomby. Możemy wpisać tam również wartości ułamkowe. Jeśli zmniejszając bombę, w każde z trzech pól – **x**, **y** i **z** – wpiszemy taką samą wartość, to nie zmienią się proporcje bomby. Jeśli skalą będzie 0.01 **D**, bomba zmniejszy się stukrotnie.



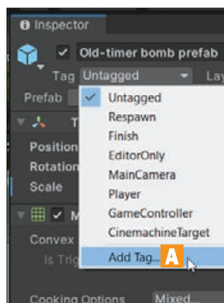


6 Zadbajmy też o pozycję bomby – tak by stała ona na terenie. O wysokości, na jakiej znajduje się bomba, mówi wartość **y** E we właściwości **Position**. Możemy edytować ją zarówno z poziomu panelu właściwości, jak i przesuwając obiekt zieloną strzałką F widoczną na obiekcie na scenie.

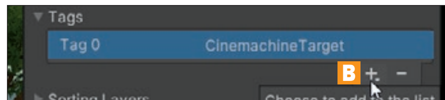
Tagowanie bomby

By odnosić się do obiektu bomby z poziomu skryptów innych obiektów, możemy nadać jej odpowiedni tag. Wykorzystanie tagów omawialiśmy już w poprzednim rozdziale tej książki.

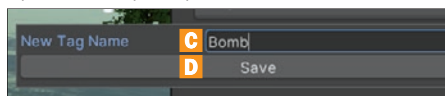
1 Jeśli rozwiniemy pole **Tag** we właściwościach bomby, zobaczymy, że nie ma tam tagu, który by do niej pasował. Tagi można jednak dodawać nie tylko z puli tagów na liście, można też tworzyć własne. Możemy więc stworzyć tag specjalnie dla bomby. W tym celu klikamy na opcję **Add Tag** A znajdującą się na samym dole listy dostępnych tagów.



2 Zostaniemy przeniesieni do panelu z tagami, w którym klikamy na plus B, by utworzyć nową pozycję na wpisanie tagu.

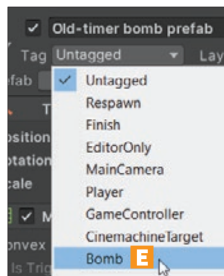


3 W niewielkim nowym oknie wpisujemy nazwę nowego tagu. Niech tag stworzony dla bomby nazywa się **Bomb** C.



4 Po podaniu nazwy należy kliknąć na przycisk **Save** D.

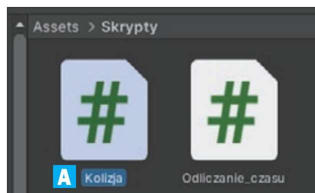
5 Wracamy do właściwości bomby i kolejny raz rozwijamy listę dostępnych tagów. Tym razem zobaczymy na niej nowo utworzony tag – **Bomb** E.



Sprawdzanie odnalezienia bomby

Kiedy bomba jest już na scenie, możemy zająć się stworzeniem skryptu, który będzie odpowiadał za sprawdzenie tego, czy gracz odnalazł bombę i czy zdążył to zrobić w wymaganym czasie.

1 Pierwszym krokiem będzie utworzenie nowego skryptu. Możemy nazwać go **Kolizja** A, choć celem jego działania nie będzie wykrywanie kolizji, jak w poprzednim rozdziale w przypadku asteroidy i statku kosmicznego, ale sprawdzanie odległości między dwoma obiektami.



2 Zakładamy, że skrypt będzie przyłączony do obiektu gracza – zatem jego lokalizację będziemy mogli sprawdzić poprzez **transform.position**. By z kolei odczytać pozycję bomby, będziemy potrzebowali w skrypcie odniesienia do odpowiedniego obiektu na scenie. W skrypcie utworzymy zatem pole reprezentujące obiekt gry, czyli bombę B.

```
public class Kolizja : MonoBehaviour
{
    public GameObject bomba;
```
B

3 Z poziomu funkcji **Start** powinniśmy stworzyć teraz połączenie między bombą, która jest na scenie, a zadeklarowanym w naszym skrypcie polem. Dzięki nadaniu

bombie na scenie tagu **Bomb** możemy przypisać obiekt ze sceny do pola poprzez odnalezienie go po nazwie tagu. Zrobimy to, pisząc w skrypcie: **bomba = GameObject.FindWithTag("Bomb");** C.

```
// Start is called before the first frame u
void Start()
{
    bomba = GameObject.FindWithTag("Bomb");
}
```

4 Sprawdzanie odległości pomiędzy graczem a bombą powinno odbywać się możliwie często – najlepiej zatem umieścić je w funkcji **Update**. Do przechowania tej odległości należy utworzyć nową zmienną o nazwie **dystans**. Jej typ to **float**, ponieważ odległość może nie być liczbą całkowitą. Do obliczenia odległości wykorzystamy funkcję **Vector3.Distance()** – cała linia kodu, jaką należy zapisać, powinna mieć postać: **float dystans = Vector3.Distance(bomba.transform.position, transform.position);** D.

5 Następnie w funkcji **Update** powinniśmy umieścić instrukcję warunkową. Poprzez nią będziemy sprawdzać, czy obliczony przez nas dystans jest mniejszy niż określony minimalny dystans, jaki możemy uznać za dojście do bomby. Jeśli dystans ten jest mniejszy niż na przykład 1.5 E, możemy w panelu konsoli wypisać komunikat o odnalezieniu bomby – by sprawdzić, czy wykrywanie jej odnalezienia zostało napisane prawidłowo.

```
// Update is called once per frame
void Update()
{
    float dystans = Vector3.Distance(b

    if (1.5f >= dystans) E
    {
        Debug.Log("Odnaleziona");
    }
}
```

6 Jeśli teraz uruchomimy rozgrywkę i odnajdziemy bombę na scenie, podchodząc do niej blisko – w oknie konsoli pojawi się napis **Odnaleziona**.

Odnalezienie bomby w określonym czasie

1 W naszym skrypcie moglibyśmy uwzględnić jeszcze jeden warunek odnalezienia bomby, czyli czas, w jakim to zrobiliśmy. By odnieść się w skrypcie, który jest przyłączony do gracza, do wartości czasu, która jest magazynowana w Stoperze, powinniśmy zadeklarować kolejne pole, które będzie serializowane (czyli **[SerializeField] public GameObject stoper;** A).

```
[SerializeField] public GameObject stoper
// Start is called before the first frame
void Start()
```

2 Po takiej zmianie skryptu we właściwościach obiektu gracza, do którego powinniśmy ten skrypt dołączyć, pojawi się już pole **Stoper**. Przeciągamy do niego obiekt Stoper z panelu **Hierarchy**.

Stoper | None (Game Object)

3 Do przechowania czasu ze stopera wykorzystamy dodatkowe pole typu **int**, które będzie prywatne.

```
[SerializeField] public GameObject stoper;
private int czas;
```

4 Jego wartość będzie nadawana w funkcji **Update**. Dostając się do wartości czasu ze stopera, musimy najpierw podać nazwę obiektu, a następnie wywołać funkcję **GetComponent**, po czym w nawiasie <> poda-

```
// Update is called once per frame
void Update()
{
D   float dystans = Vector3.Distance(bomba.transform.position, transform.position);
```

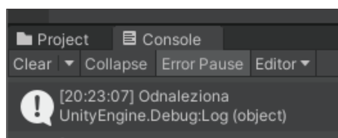
jemy nazwę komponentu – która w tym wypadku jest nazwą naszego skryptu odliczającego czas. Następnie po zwykłym nawiasie stawiamy kropkę i podajemy nazwę wartości, którą chcemy odczytać. Całość polecenia powinna mieć zatem postać: **czas = stoper.GetComponent< Odliczanie_czasu>().czas;** B.

```
void Update()
{
    float dystans = Vector3.Distance(bomba.transform.positi
B   czas = stoper.GetComponent<Odliczanie_czasu>().czas;
    if (1.5f >= dystans)
```

5 Gdy czas jest już odczytany, możemy w istniejącej już w funkcji instrukcji warunkowej dodać drugi warunek – czas powinien być większy niż 0 C.

```
    czas = stoper.GetComponent<Odliczar
    if (1.5f >= dystans && czas > 0) C
    {
        Debug.Log("Odnaleziona");
    }
```

6 Po takich zmianach informacja o odnalezieniu bomby będzie się pojawiać w oknie konsoli tylko wtedy, gdy podejdziemy do bomby przed upłynięciem określonego czasu.



Eksplozja bomby

Kolejnym elementem realizacji naszej gry będzie dodanie na scenie wybuchu – gdy czas na odnalezienie bomby upłynie, a ta nie zostanie w tym czasie jeszcze odnaleziona. Do obsługi eksplozji wykorzystamy nasz skrypt stworzony na potrzeby stopera. To w nim najłatwiej będzie sprawdzić, czy upłynął czas przewidziany na poszukiwanie bomby.

1 Pierwsza rzecz, jaką należy zmodyfikować w skrypcie stopera, to dodanie dwóch nowych pól – pierwsze z nich powinno mieć typ **GameObject** A i pozwalać na nadanie mu wartości poprzez panel właściwości edytora Unity. To w nim umieścimy prefabrykat wybuchu.

```
[SerializeField] public GameObject wybuch; A
// Start is called before the first frame update
public string stan_rozgrywki = "szukanie"; B
private float odliczony = 0f;
void Start()
{
```

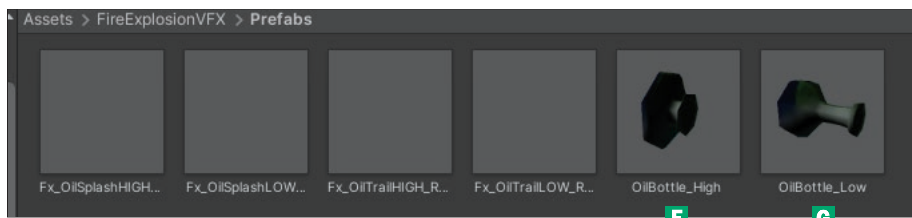
C# więcej na stronie 33

2 Drugie z potrzebnych nam pól będzie typu łańcuchowego – zapiszemy tam tekstową informację o tym, jaki jest stan rozgrywki. Informacja ta będzie nam potrzebna do tego, by wiedzieć, czy mamy dalej odliczać czas i by móc odpowiednio zareagować na upłynięcie czasu i odnalezienie bomby. Początkowa wartość tego pola to **szukanie** B. Gdy **stan_rozgrywki** wciąż będzie mieć taką wartość, to znaczy, że czas nie upłynął, a bomba nie została odnaleziona.

3 Instrukcje znajdujące się dotychczas w funkcji **Update** powinny być wykonywane tylko wtedy, kiedy wciąż trwa szukanie bomby – zatem powinniśmy stworzyć na samym początku instrukcję warunkową, w której warunku sprawdzimy, czy **stan_rozgrywki** ma wartość **szukanie** C. Jeśli tak jest, powinny się wykonać wszystkie instrukcje, które do tej pory były w funkcji.

```
void Update()
{
    if (stan_rozgrywki == "szukanie") C
    {
        odliczony = odliczony + Time.deltaTime;
        if (odliczony >= 1f)
        {
            czas = czas - 1;
            odliczony = 0f;
        }
        t.text = "Czas do końca: " + czas.ToString();
    }
}
```

4 By czas przestał się odmierzać, należy sprawdzić, czy upłynął on już do końca. Zatem wewnątrz dodanej w poprzednim kroku instrukcji warunkowej powinniśmy na końcu dodać kolejną instrukcję warunkową,

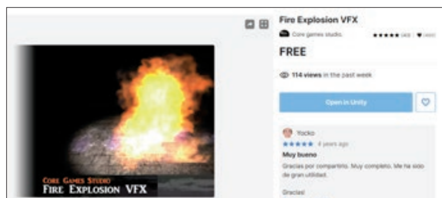


w której sprawdzimy, czy **czas** jest mniejszy od 0, czy równy 0 D.

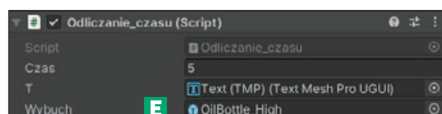
```
t.text = "Czas do końca: " + czas.ToString();
if (czas <= 0) D
{

}
}
}
```

5 Gdy podany warunek jest spełniony, bomba powinna wybuchnąć, a czas przestać się odliczać. By zrealizować pierwsze z tych zadań, powinniśmy połączyć prefabrykat wybuchu z polem na wybuch we właściwościach stopera. W naszym przykładowym projekcie została dodana paczka zasobów **Free Explosion VFX**.



6 Jej zawartość w zasobach projektu jest umieszczona w folderze **FreeExplosionVFX**, a prefabrykaty obiektów dających wybuch znajdują się w będącym w nim folderze **Prefabs**. W pole na wybuch we właściwościach **stopera** E powinien znaleźć się jeden z dwóch ostatnich prefabrykatów w folderze – **OilBottle_High** F lub **OilBottle_Low** G.



7 Wracamy do skryptu – w utworzonej przez nas instrukcji warunkowej powinniśmy teraz sprawić, że obiekt ze slotu we właściwościach pojawi się na scenie. Do tego służy instrukcja **Instantiate**, która tym razem może mieć postać **Instantiate(wybuch, GameObject.FindWithTag("Bomb").transform.position, Quaternion.identity);** H, gdzie **GameObject.FindWithTag("Bomb").transform.position** odpowiada za pozycję pojawienia się wybuchu i jest to pozycja, w której jeszcze znajduje się bomba. Taki zapis również pozwala nam na odczytanie pozycji innego obiektu ze sceny bez deklarowania pola na jego przechowanie w skrypcie.

8 By wybuch był lepiej widoczny, bomba może zniknąć. Nie powinniśmy jej jednak usuwać ze sceny, tylko przesunąć pod **Terrain**, czyli zmienić jej pozycję. I taka instrukcja przesunięcia obiektu z bombą w dół powinna znaleźć się dalej w skrypcie. Może mieć ona postać: **GameObject.FindWithTag("Bomb").transform.position += Vector3.down *10;** I – ponownie odwołujemy się do bomby poprzez nadany jej wcześniej tag i zmieniamy pozycję, dodając do niej wektor przesunięcia do dołu. By przesunięcie było większe, wektor ten jest pomnożony razy 10.

```
if (czas <= 0)
{                                                                                    H
    Instantiate(wybuch, GameObject.FindWithTag("Bomb").transform.position, Quaternion.identity);
```

```
Instantiate(wybuch, GameObject.FindWithTag("Bomb").transform.position, Quaternion.identity);
GameObject.FindWithTag("Bomb").transform.position += Vector3.down *10; I
```

```
if (czas <= 0)
{
    Instantiate(wybuch, GameObject.FindWithTag("Bomb").transform.position, Quaternion.identity)
    GameObject.FindWithTag("Bomb").transform.position += Vector3.down *10;
    stan_rozgrywki = "przegrana"; J
}
```

9 Ostatnia instrukcja, jaką należy wykonać w tym bloku w instrukcji warunkowej, to zmiana wartości pola **stan_rozgrywki**, która teraz powinna być równa napisowi "przegrana" **J**.

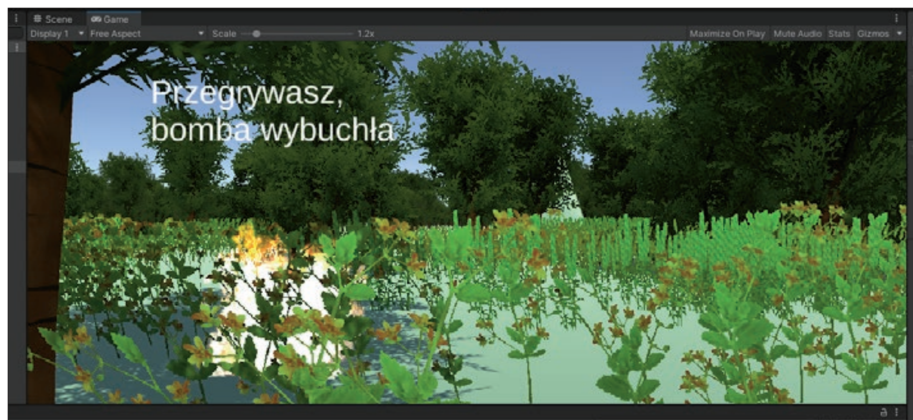
10 Wewnątrz funkcji **Update**, po sprawdzeniu tego, czy **stan_rozgrywki** ma wartość **szukanie**, powinniśmy umieścić kolejną instrukcję warunkową, poprzez którą sprawdzimy, czy **stan_rozgrywki** ma wartość **przegrana**.

```
            stan_rozgrywki = "przegrana";
        }
    }
    if (stan_rozgrywki == "przegrana")
    {
```

11 Jeśli warunek jest spełniony, powinniśmy zmienić tekst na obiekcie tekstowym, by nie wyświetlał on już czasu, który i tak nie będzie się dalej odliczał, ale aby pojawiła się informacja o przegranej. Zrobimy to, umieszczając w kodzie instrukcję: **t.text = "Przegrywasz, bomba wybuchła";** **K**.

```
if (stan_rozgrywki == "przegrana")
{
  K t.text = "Przegrywasz, bomba wybuchła";
}
```

12 Możemy teraz przetestować grę – jeśli nie podejdziemy do bomby na czas, bomba zniknie, a w jej miejscu pojawi się eksplozja. Dodatkowo w oknie gry będzie widoczny komunikat o przegranej.



Wygrana

Do zakończenia tworzenia naszej gry pozostało nam jeszcze zaprogramowanie wygranej. Mamy już stworzony system wykrywania, czy doszliśmy do bomby – trzeba jedynie nieco go zmienić, by nie wyświetlał informacji w konsoli o odnalezieniu bomby,

C# więcej na stronie 35

```
if (1.5f >= dystans && czas > 0)
{
    stoper.GetComponent<Odliczanie_czasu>().stan_rozgrywki = "wygrana"; A
}
```

ale ją „rozbroił", czyli zatrzymał odliczanie czasu do wybuchu.

1 Pierwszym krokiem będzie edycja skryptu **Kolizja**, w którym wykrywane jest dojście do bomby. Polecenie **Debug.Log** powinno zostać zastąpione przez **stoper.GetComponent<Odliczanie_czasu>().stan_rozgrywki = "wygrana";** A, co nada polu **stan_rozgrywki** ze skryptu stopera zupełnie nową wartość, która jeszcze nie jest przez ten skrypt obsługiwana.

2 W skrypcie stopera powinniśmy umieścić kolejną instrukcję warunkową, sprawdzającą następny możliwy stan rozgrywki B.

```
    t.text = "Przegrywasz, bomba wybuchła";
}
if (stan_rozgrywki == "wygrana") B
{

}
```

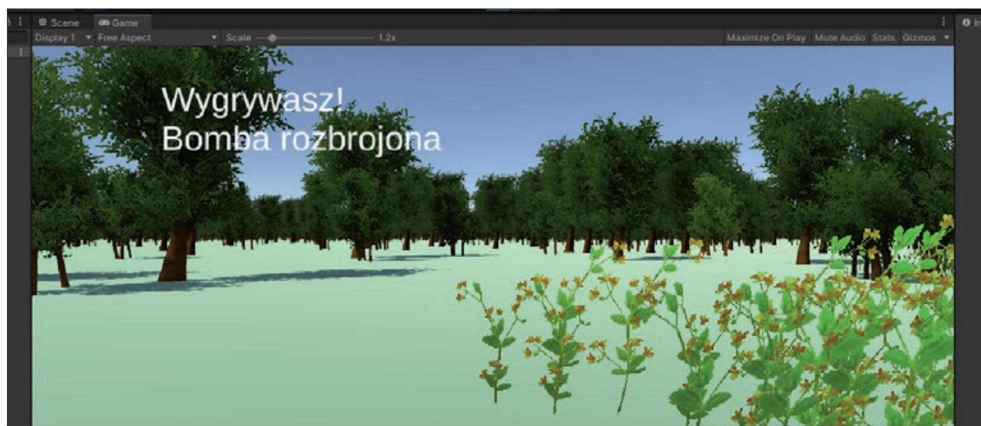
3 Gdy **stan_rozgrywki** ma wartość **wygrana**, powinniśmy informację o wygranej C umieścić na obiekcie tekstowym, na którym wcześniej wyświetlano czas pozostały do wybuchu bomby.

```
if (stan_rozgrywki == "wygrana")
{                                                C
    t.text = "Wygrywasz! Bomba rozbrojona";
```

4 Podobnie jak w przypadku przegranej, tak i tu możemy przenieść bombę pod teren, by nie było jej już widać – kolejny raz umieszczając w kodzie linię: **GameObject.FindWithTag("Bomb").transform.position += Vector3.down *10;** D.

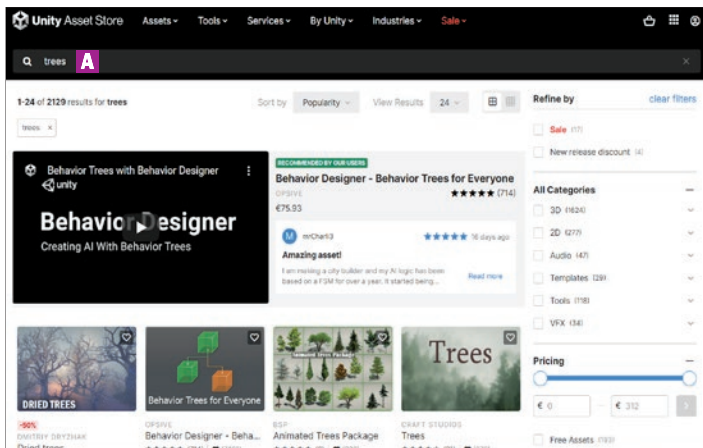
5 Tak stworzoną grę można już testować. Gracz może w niej już nie tylko przegrać, ale też wygrać. Jeśli odnajdziemy bombę przed upłynięciem ustawionego czasu, zobaczymy w oknie rozgrywki komunikat o rozbrojeniu bomby i wygranej.

```
t.text = "Wygrywasz! Bomba rozbrojona";                    D
GameObject.FindWithTag("Bomb").transform.position += Vector3.down * 10;
```



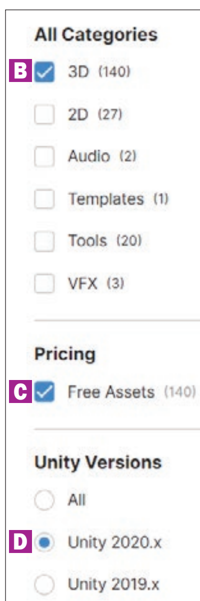
Inne zasoby prefabrykatów

Do budowy lasu w tym rozdziale wykorzystana została paczka zasobów **Nature Starter Kit 2**. Nie jest to oczywiście jedyna paczka zasobów zawierająca drzewa, jaką można wykorzystać w projekcie.

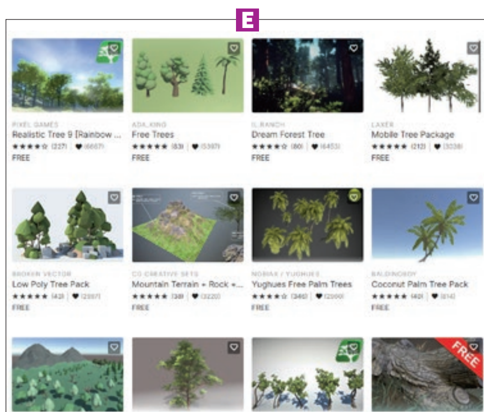


1 Jeśli w Asset Store chcemy znaleźć zasoby zawierające drzewa, zamiast konkretnej nazwy zestawu wpiszmy **trees** w pasku wyszukiwania **A**. Wyświetlona zostanie cała lista różnych paczek zawierających drzewa.

2 Lista zawiera też pozycje płatne, a także takie, które mogą nie być zgodne z wersją Unity wykorzystywaną w naszym projekcie. Panel po prawej stronie okna Asset Store pozwala na filtrowanie wyników. By wyświetlić tylko darmowe zasoby, jakie spełniają wymagania naszego projektu, należy w sekcji **All Categories** zaznaczyć opcję **3D** **B**. W sekcji **Pricing** zaznaczamy **Free Assets** **C**, a w sekcji **Unity Versions** wybieramy **Unity 2020.x** **D**.



3 Przy tak zmienionym zakresie poszukiwań dostaniemy zestawienie paczek **E**, jakie można importować do projektu zamiast wcześniej użytych drzew i wstawiać na obiekt **Terrain** poprzez opcję zadrzewiania.



NA KONIEC: EKSPORT

Podobnie jak w przypadku projektu opisanego w poprzednim rozdziale – gotową grę możemy eksportować do postaci wykonywalnej, korzystając z opcji **Build And Run** dostępnej w pozycji **File** menu górnego.

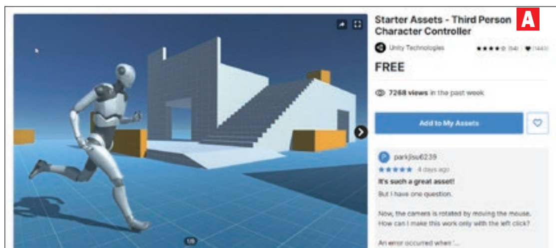


7 Gra z widokiem trzecioosobowym

W tym rozdziale stworzymy trójwymiarową grę platformową. Zadaniem gracza będzie w niej zebranie wszystkich umieszczonych na mapie diamentów w taki sposób, by nie spaść z platformy i wykonać zadanie w jak najkrótszym czasie

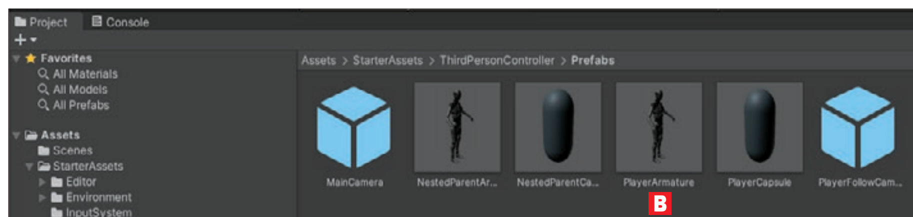
Kontroler postaci z widokiem trzecioosobowym

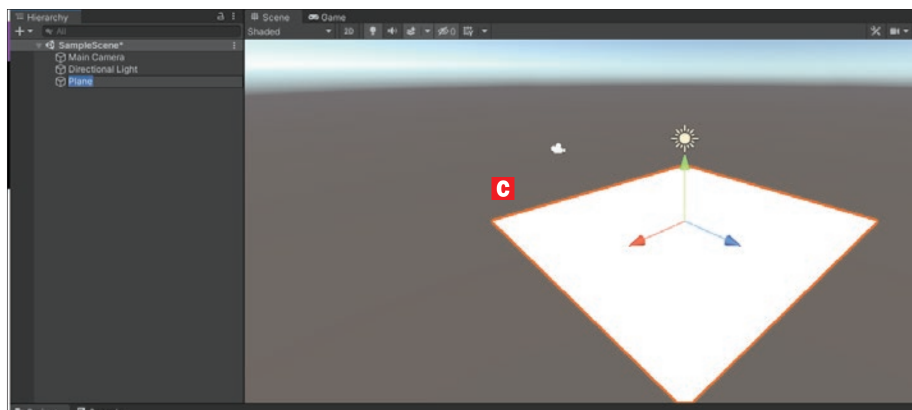
Do stworzenia tego typu gry potrzebny będzie nam kontroler postaci z widokiem trzecioosobowym. W naszej grze kamera cały czas będzie pokazywała postać, którą gracz porusza się po scenie. Zatem kamera – podobnie jak w grze z widokiem pierwszoosobowym – też będzie musiała poruszać się wraz z graczem.



1 Cały ten mechanizm, a nawet więcej, możemy uzyskać, umieszczając w naszym projekcie paczkę zasobów **Starter Assets – Third Persor Character Controller** A.

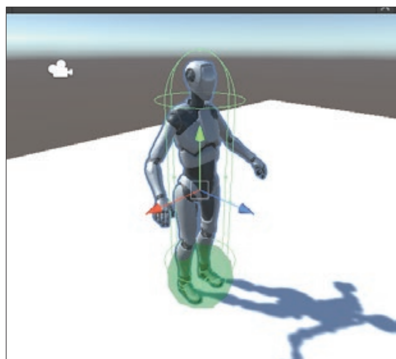
2 W paczce tej znajduje się kilka prefabrykatów, między innymi **PlayerArmature** B – czyli prefabrykat humanoidalnego robota, który może być postacią w naszej grze.



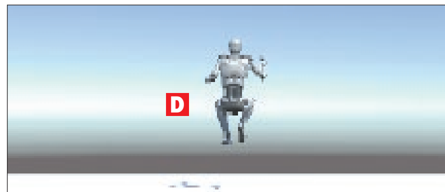


3 Zanim ustawimy postać na scenie, należy wygenerować choć fragment mapy gry. Inaczej nasz robot nie będzie miał po czym się poruszać. Dlatego umieszczamy na scenie płaszczyznę **C** (obiekt **Plane** z grupy **3D Object**).

4 Na nią z importowanej paczki zasobów przeciągamy **PlayerArmature**.



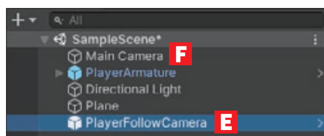
5 Możemy już teraz uruchomić podgląd rozgrywki. Umieszczona na scenie postać może się poruszać (sterowana jest strzałkami lub klawiszami W A S D), a nawet skakać **D**. Postać może także biegać. Bieg uruchamiamy, „chodząc" postacią z wciśniętym klawiszem shift.



Ruch kamery za postacią

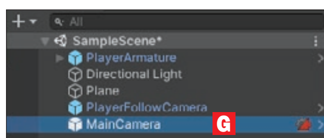
Postać porusza się po płaszczyźnie, jednak nie porusza się wraz z nią kamera. Musimy temu zaradzić.

1 Wśród prefabrykatów w pobranej paczce znajduje się taki o nazwie **PlayerFollowCamera** **E**. Dodajemy go na scenę. To obiekt obsługujący ruch kamery.

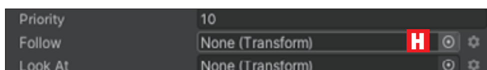


2 By ruch kamery był poprawnie obsłużony, trzeba usunąć ze sceny domyślną kamerę, czyli obiekt **Main Camera** **F**.

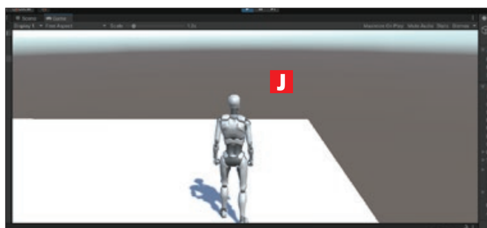
3 Dodajemy na scenę nieco zmodyfikowaną kamerę z puli prefabrykatów z pobranej paczki zasobów – prefabrykat o nazwie **MainCamera** **G**.

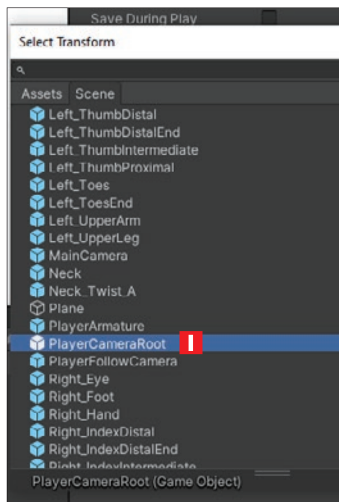


4 By kamera poruszała się we właściwy sposób, obiekt obsługujący ruch kamery wymaga od nas określenia w swoich właściwościach, za czym ma podążać. Obiekt do śledzenia wybieramy w polu **Follow** **H** właściwości. Klikamy na kółko po prawej stronie slotu na obiekt.



5 Wyświetlona zostanie lista obiektów na scenie, które można wybrać. Tu należy posłużyć się obiektem **PlayerCameraRoot** **I**,



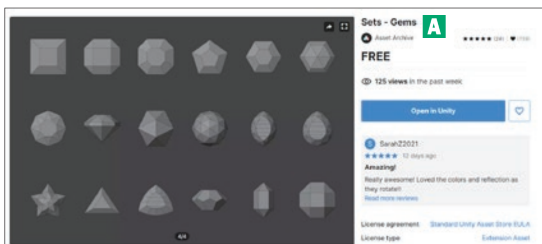


który jest jednym z elementów budujących naszego humanoidalnego robota.

6 Przy ponownym uruchomieniu rozgrywki **J** zobaczymy, że kamera jest już zupełnie inaczej ustawiona i podąża za postacią.

Diamenty

Zgodnie z założeniami naszej gry postać, poruszając się po mapie, powinna jak najszybciej zbierać umieszczone na niej diamenty. W tym kroku zajmiemy się dodaniem diamentów do projektu i zaprogramowaniem ich zbierania.



1 Jako diamenty możemy wykorzystać prefabrykaty trójwymiarowych obiektów z paczki zasobów **Sets – Gems** **A**.

2 Po imporcie w zasobach projektu pojawi się folder o nazwie takiej, jak nazwa paczki zasobów. W nim znajduje się katalog **Prefabs**, a w nim kolejne foldery **B**, w których nazwach pojawiają się kolory. Dopiero po wybraniu diamentów w kolorze, który nam się podoba, zobaczymy pełne zestawie-





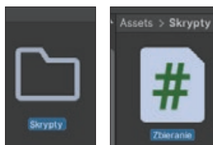
nie prefabrykatów diamentów, które będą miały odpowiedni kolor.

3 Wybieramy jeden z prefabrykatów diamentów z grupy o dowolnym kolorze i umieszczamy go na scenie **C**.

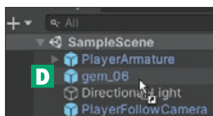
Zbieranie diamentów

1 Do obsługi zbierania diamentów na scenie wykorzystamy nowy skrypt. Ponieważ nie tworzyliśmy w tym projekcie jeszcze własnych skryptów, tworzymy najpierw folder **Skrypty**, a następnie w nim zapisujemy skrypt o nazwie **Zbieranie**.

2 Utworzony skrypt przeciągamy na diament **D** w panelu **Hierarchy**.



3 W utworzonym skrypcie usuwamy znajdujące się tam funkcje **Start** i **Update**.




```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Zbieranie : MonoBehaviour
{

}
```

4 Zamiast nich dodajemy funkcję **OnTriggerEnter** **E**, która będzie wywoływana w momencie wejścia obiektu na diament. Funkcja wywoływana przez takie zdarzenie powinna mieć parametr typu **Collider** **F**, który w obrębie naszej funkcji będziemy nazywać **other** **G**.

5 Nasza funkcja powinna reagować zniknięciem diamentu, ale tylko wtedy, gdy obiektem, z którym koliduje diament, jest postać gracza. Dlatego wewnątrz utworzo-

```
public class Zbieranie : MonoBehaviour
{
    void OnTriggerEnter(Collider other)
    {

    }
}
```

E F G

nej funkcji powinna znaleźć się instrukcja warunkowa sprawdzająca, czy to gracz jest obiektem, który wszedł na diament. Warunek w tej instrukcji może brzmieć **other.tag == "Player"** **H**. W tym wypadku możemy sprawdzić obiekt przez tag, nawet pomimo tego, że nie nadawaliśmy tagu temu obiektowi – tag **Player** jest w nim ustawiony domyślnie, już w prefabrykacie.

```
void OnTriggerEnter(Collider other)
{
    if(other.tag == "Player")
    {

    }
}
```

H

C# więcej na stronie 36

6 Wewnątrz instrukcji warunkowej możemy określić, co ma się stać, gdy obiekt gracza wejdzie w diament. Jeśli zapiszemy tam **Destroy(gameObject);** **I**, usuniemy ze sceny ten obiekt, do którego przypięto wykonywany skrypt, czyli diament.

```
if(other.tag == "Player")
{
    Destroy(gameObject);
}
```

I

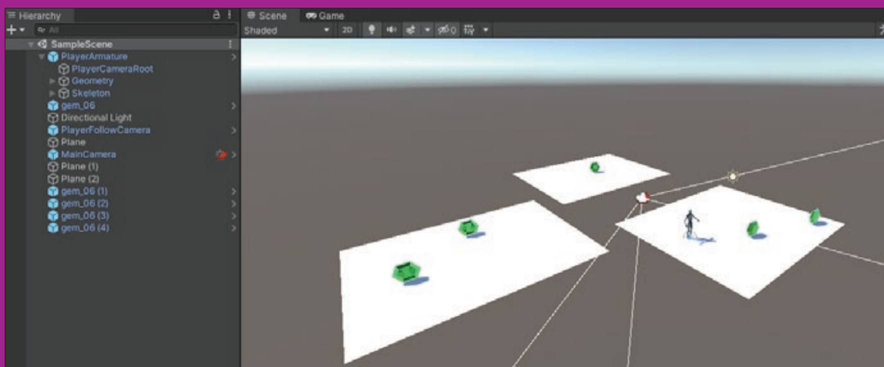
7 By całość skryptu działała poprawnie, obiekt będący diamentem musi mieć **collider** – do wykrywania wejścia w niego przez postać gracza. Dodajemy zatem kom-

ROZBUDOWANIE MAPY

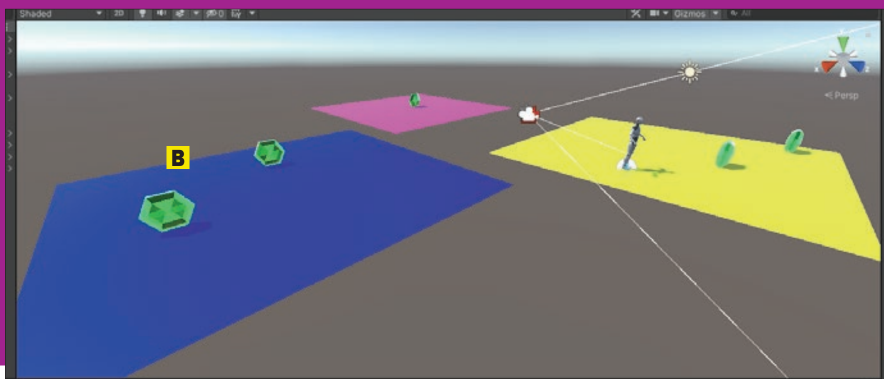
Klikając na obiekty w panelu **Hierarchy** prawym przyciskiem myszy, rozwijamy ich menu kontekstowe, w którym jedną z opcji jest **Duplicate**, czyli tworzenie kopii obiektów. Korzystając z tej opcji, stwórzmy nowe płaszczyzny i diamenty. Poustawiajmy je tak, by pomiędzy płaszczyznami była pewna odległość i by trzeba było przeskakiwać między nimi, aby na nie wejść. Diamenty powinny być poustawiane nad różnymi płaszczyznami. W ten sposób tworzymy kolejne platformy w naszej grze platformowej.

Rozmieszczając nowe płaszczyzny na scenie, możemy zmieniać ich wymiary – nie każda platforma musi być taka sama. Ważne jednak, aby nie zmieniać wysokości, na jakiej się one znajdują. Zatem upewniamy się we właściwościach płaszczyzn, że w grupie **Transform** właściwość **Position** ma w polu **Y** wartość **0** **A**. By zadbać o wygląd gry, możemy jeszcze zmienić materiały, jakimi pokryte są platformy **B**.

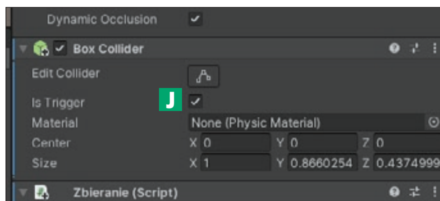
Taki materiał można utworzyć w zasobach projektu jako zupełnie nowy materiał lub na podstawie tekstury – jak w przypadku projektu ze statkiem kosmicznym, gdy przypisywaliśmy teksturę tła do płaszczyzny pod statkiem.



Przykładowe rozmieszczenie płaszczyzn i diamentów na scenie



ponent **Box Collider** do diamentu i w dodanym komponencie aktywujemy jeszcze opcję **Is Trigger** J.

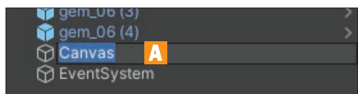


8 Po uruchomieniu podglądu rozgrywki postać będzie mogła już zebrać znajdujący się na mapie diament.

Obiekty tekstowe

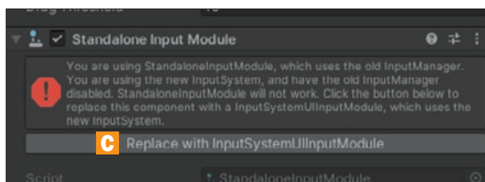
Dla gracza w naszej grze istotne będą dwie informacje. Pierwsza z nich to czas, w jakim uda mu się przejść aż do mety, a druga to liczba zebranych przy tym diamentów. Te dwie informacje powinny być stale widoczne w oknie gry, dlatego przygotujemy dwa obiekty tekstowe do ich wyświetlania.

1 By umieścić w oknie gry obiekty tekstowe, niezbędne jest najpierw dodanie obiektu **Canvas** A do sceny.

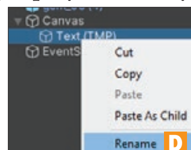


2 Następnie dodajemy obiekt tekstowy **Text – TextMeshPro**, po wyborze którego program wyświetli nam kolejne okno. W nim musimy importować do projektu paczkę zasobów obsługujących obiekt tekstowy, klikając na **Import TMP Essentials** B.

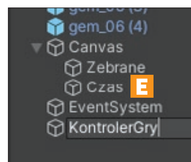
3 Program będzie generował jednak błąd, którego pozbycie będzie możliwe z poziomu właściwości obiektu **EventSystem**, który został dodany automatycznie do sceny wraz z dodaniem obiektu **Canvas**. Wybieramy przycisk **Replace with InputSystemUIInputModule** C.



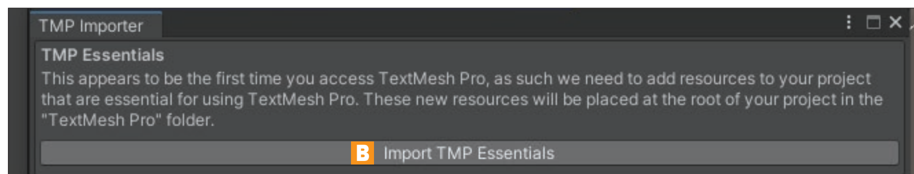
4 Będą nam potrzebne dwa obiekty tekstowe w oknie gry. Aby móc rozróżniać te obiekty na scenie, każdy z nich powinien mieć inną nazwę. Zmieńmy nazwę dodanego już obiektu tekstowego na **Zebrane**. By móc to zrobić, klikamy prawym przyciskiem myszy na obiekt w panelu **Hierarchy** i wybieramy z wyświetlonego menu kontekstowego opcję **Rename** D.

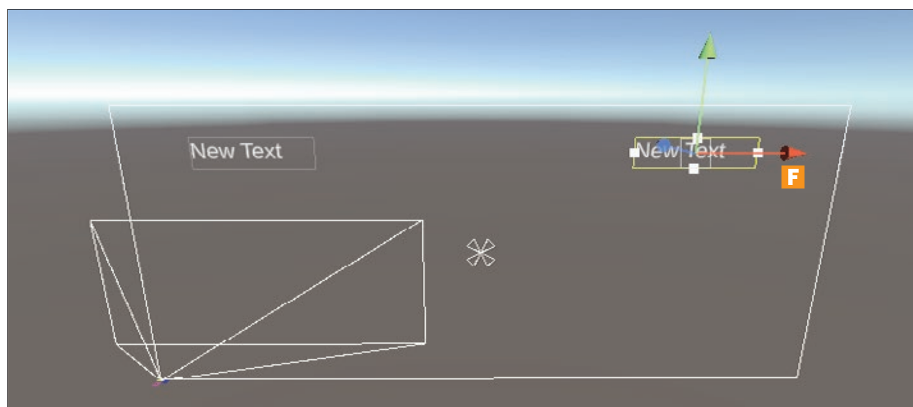


5 Dodajemy jeszcze jeden obiekt tekstowy E na **Canvasie**. Jemu również zmieniamy nazwę – tym razem na **Czas**.

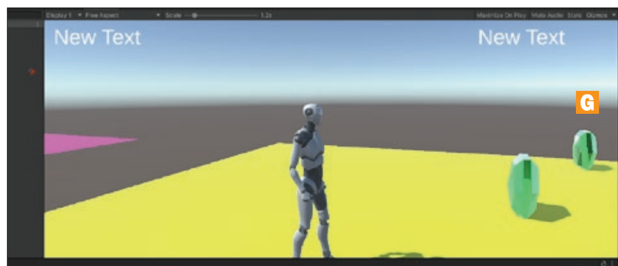


6 Rozmieszczamy dwa obiekty tekstowe w taki sposób, by każdy z nich znajdował się w innym rogu okna. Do przesuwania





obiektów możemy użyć kolorowych strzałek F lub panelu właściwości. Zmieniając współrzędne napisów, postarajmy się, aby były one na tej samej pozycji **Y**, co zapewni obydwu napisom taką samą odległość od górnej krawędzi okna.



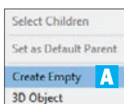
7 Sprawdzamy jeszcze, jak obiekty tekstowe prezentują się w oknie gry w trybie rozgrywki G.

8 Początkowa wartość zapisana w obiektach tekstowych nie musi być zmieniana – i tak zmienimy ją z poziomu skryptów.

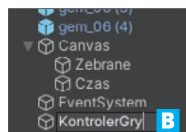
Kontroler gry

Na naszej scenie powinien pojawić się obiekt, który wykorzystamy do zarządzania całą rozgrywką. W przypadku poprzedniego projektu stworzyliśmy obiekt Stoper, który w pewnym sensie pełnił podobną rolę. Takie obiekty zarządzające stanem rozgrywki i magazynujące w sobie pewne wartości, które są zmieniane i wczytywane przez inne obiekty na scenie, nazywamy kontrolerami.

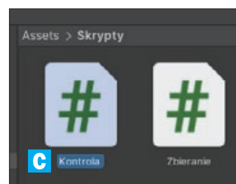
1 By stworzyć kontroler, otwieramy w panelu **Hierarchy** nowy obiekt sceny, wybierając **Create Empty** A.



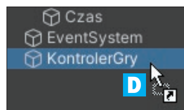
2 Zmieniamy temu obiektowi nazwę na **KontrolerGry** B.



3 Jedynym zadaniem tego obiektu będzie wykonywanie dołączonego do niego skryptu kontrolującego całą rozgrywkę – to znaczy odliczającego czas, zliczającego zebrane diamenty czy odpowiednio reagującego na wygranie i przegranie gry. Tworzymy w zasobach



projektu skrypt o nazwie **Kontrola** C i przeciągamy go na obiekt **KontrolerGry** D.



Zliczanie zebranych diamentów

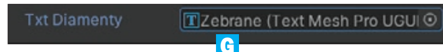
Pierwszym zadaniem wykonywanym przez nasz kontroler gry będzie zliczanie zebranych przez gracza diamentów. Znikają one w momencie, gdy wejdzie na nie postać gracza, ale nie mamy w programie w żaden sposób przechowywanej sumy zebranych diamentów.

1 Liczba zebranych diamentów powinna być przechowywana w odpowiednim polu w skrypcie **Kontrola**. Przed definicjami funkcji, jakie są domyślnie umieszczone w skrypcie, dopisujemy: **public int zebrane_diamenty = 0;** E. Utworzymy w ten sposób pole o nazwie **zebrane_diamenty**, w którym będziemy mogli zapisywać liczby całkowite.

```
public class Kontrola : MonoBehaviour
{
  E public int zebrane_diamenty = 0;
    // Start is called before the first
    void Start()
```

2 Drugim polem, jakie dodamy do skryptu, powinno być pole typu **TextMeshProUGUI** F, które możemy nazwać txtDiamenty. Pole powinno być utworzone w taki sposób, by wygenerowało nowy slot we właściwościach obiektu (należy dodać **[SerializeField]**, co sprawi, że wszystkie pola oznaczone jako **public** będą generowały sloty w panelu właściwości w edytorze Unity). W ten slot będziemy mogli wstawić obiekt tekstowy o nazwie **Zebrane** G dodany przez nas na potrzeby wyświetlania liczby zebranych diamentów.

```
    [SerializeField]
  F public TextMeshProUGUI txtDiamenty;
    // Start is called before the first
    void Start()
    {
```

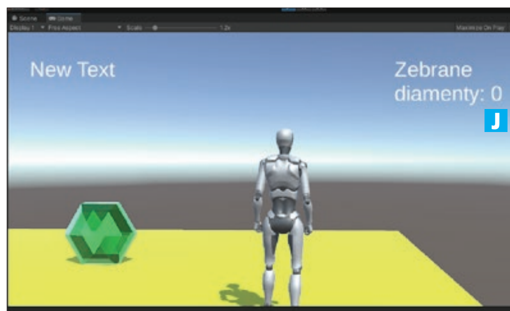


3 By móc korzystać w skrypcie z pól typu **TextMeshProUGUI**, skrypt powinien dostać przestrzeń nazwy **TMPro** H.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using TMPro;  H
```

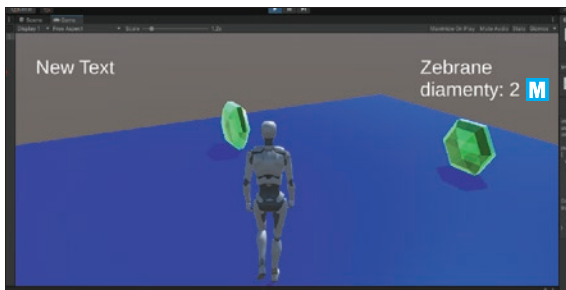
4 By tekst na obiekcie zmieniał się na bieżąco, należy jego zmianę umieścić w funkcji **Update**. Na obiekcie tekstowym powinien pojawić się fragment tekstu opisujący, co oznacza liczba, którą dopiszemy do tego obiektu tekstowego. By pokazać liczbę zebranych diamentów, powinniśmy zapisać linię kodu: **txtDiamenty.text = "Zebrane diamenty: " + zebrane_diamenty.ToString();** I.

5 Jeśli uruchomimy teraz podgląd gry, to zamiast napisu **New Text** na jednym z obiektów tekstowych będzie znajdowała się już informacja o zebranych diamentach. Jednak ich liczba wciąż będzie pokazana jako **0** J.

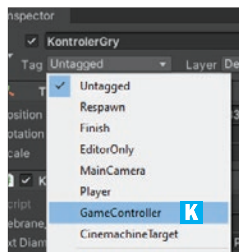



```
void Update()
{
  I txtDiamenty.text = "Zebrane diamenty: " + zebrane_diamenty.ToString();
}
```

6 By temu zaradzić, należałoby zmodyfikować wartość pola **zebrane_diamenty**. Ta modyfikacja powinna jednak odbywać się w momencie zebrania diamentu przez gracza. Ten moment jest wychwytywany przez skrypt o nazwie **Zbieranie**, który dołączony jest do obiektów przedstawiających diamenty.



Mamy oczywiście możliwość zmiany w tamtym skrypcie wartości pola ze skryptu innego obiektu – jednak w skrypcie **Zbieranie** powinno wtedy istnieć odniesienie do naszego kontrolera gry. Takie odniesienie może być na przykład wyłapaniem obiektu poprzez jego tag. Jednak by można było odnieść się do kontrolera gry poprzez tag, obiekt ten powinien mieć nadany jakiś tag. Możemy to zrobić z poziomu panelu właściwości kontrolera naszej gry. Na liście dostępnych tagów jest nawet pozycja **GameController** K, która jest wykorzystywana właśnie do tagowania kontrolerów gry.



7 Gdy kontroler gry ma tag, możemy edytować skrypt **Zbieranie**. Skrypt oprócz polecenia **Destroy(gameObject);** powinien wykonać jeszcze inną operację w momencie wejścia gracza na diament. Tą operacją będzie **GameObject.FindWithTag("GameController").GetComponent<Kontrola>().zebrane_diamenty += 1;** L – to wychwytywanie obiektu poprzez tag, a następnie wybieranie z jego komponentów komponentu o nazwie **Kontrola** i wczytywanie z niego pola **zebrane_diamenty**. Poprzez zapis **+= 1** zwiększa się o 1 przechowywaną tam liczbę.

8 Jeśli teraz przetestujemy rozgrywkę, zobaczymy, że wraz ze zbieraniem kolejnych diamentów ich liczba M zmienia się także w oknie gry.

Odliczanie czasu gry

Kolejnym z elementów obsługiwanych przez nasz kontroler gry powinno być odliczanie czasu. Podobnie jak w przypadku liczby diamentów – czas będzie musiał być zadeklarowany w skrypcie. Potrzebne będzie też nawiązanie połączenia pomiędzy skryptem a obiektem dedykowanym do wyświetlania czasu.

1 Należy utworzyć pole klasy o nazwie **czas_gry**. Pole może przechowywać liczby całkowite i mieć wartość początkową równą 0 A.

```
public TextMeshProUGUI txtDiame
A private int czas_gry = 0;
  // Start is called before the f
  void Start()
```

2 By dodać w skrypcie odwołanie do obiektu tekstowego stworzonego do wyświetlania czasu, jaki upłynął od startu rozgrywki, napiszemy **public TextMeshProUGUI txtCzas;** B.

```
[SerializeField]
public TextMeshProUGUI txtDiamenty
public TextMeshProUGUI txtCzas; B
```

```
if(other.tag == "Player")
{
    Destroy(gameObject);
  L GameObject.FindWithTag("GameController").GetComponent<Kontrola>().zebrane_diamenty += 1;
}
```

3 Czas rozgrywki powinien być liczony tylko, gdy rozgrywka trwa. Kiedy przegrywamy i wygrywamy – dalsze liczenie czasu nie jest nam potrzebne. By odpowiednio określić moment liczenia czasu, dodajemy jeszcze pole typu **string** o nazwie **stan_gry** i wartości początkowej **trwa** **C**.

```
private int czas_gry = 0;
public string stan_gry = "trwa"; C
// Start is called before the fi
```

4 Czas gry będziemy odliczać co sekundę. Dodajemy jeszcze jedno pole klasy, w którym będziemy przechowywać upływający czas aż do upłynięcia pełnej sekundy i dopiero, gdy zmienna ta dostanie wartość na to wskazującą, zwiększymy o 1 wartość pola z czasem gry. Tworząc nowe pole, nadajemy mu typ **float** **D**.

```
public string stan_gry = "trwa";
private float sekunda = 0f; D
```

5 W funkcji **Update** mamy już zapisaną zmianę w oknie gry wartości zebranych diamentów. Z czasem będzie podobnie – jednak nowy zmieniony czas powinien pokazywać się w oknie tylko wtedy, kiedy gramy. Trzeba zatem w skrypcie umieścić instrukcję warunkową, poprzez którą sprawdzimy, czy wartość pola **stan_gry** jest równa **trwa**.

6 Jeśli warunek jest spełniony, powinniśmy powiększyć wartość pola **sekunda** o czas, jaki upłynął od wyświetlenia ostatniej klatki rozgrywki, co uzyskamy, zapisując: **sekunda += Time.deltaTime;** **E**.

```
if (stan_gry == "trwa")
{
    sekunda += Time.deltaTime; E

}
```

7 Dalej powinniśmy sprawdzić, czy zmagazynowana w tym polu wartość osiągnęła już co najmniej 1 **F**, co zrobimy, dodając nową instrukcję warunkową.

```
sekunda += Time.deltaTime;
if (sekunda >= 1f) F
{

}
```

8 Pierwszym krokiem po odliczeniu sekundy może być wyzerowanie pola **sekunda** **G**, tak by pozwalało ono na odliczenie kolejnej sekundy.

```
if (sekunda >= 1f)
{
G   sekunda = 0;
}
```

9 Drugim krokiem po odliczeniu sekundy powinna być edycja wartości pola **czas_gry** **H** poprzez zwiększenie jej o 1.

```
if (sekunda >= 1f)
{
    sekunda = 0;
H   czas_gry += 1;
}
```

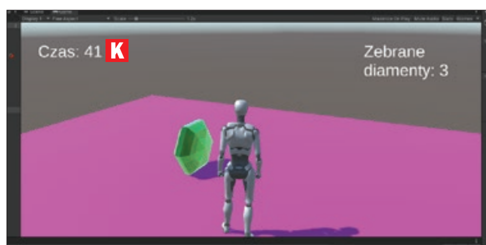
10 Po zakończeniu instrukcji warunkowej moglibyśmy dodać wypisanie odliczonego czasu na obiekcie tekstowym, co uzyskamy, pisząc w skrypcie: **txtCzas.text = "Czas: " + czas_gry.ToString();** **I**.

```
        czas_gry += 1;
    }
I   txtCzas.text = "Czas: " + czas_gry.ToString();
}
```

11 By móc przetestować odliczanie czasu, obiekt tekstowy **Czas** **J** musi znaleźć się w nowym slocie, jaki powinien pojawić się już we właściwościach kontrolera gry.

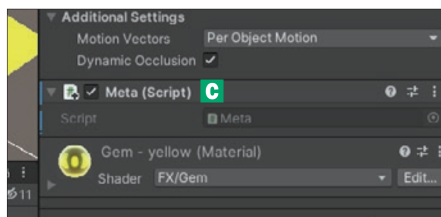
Txt Diamenty	Zebrane (Text Mesh Pro UGUI)
Txt Czas **J**	Czas (Text Mesh Pro UGUI)

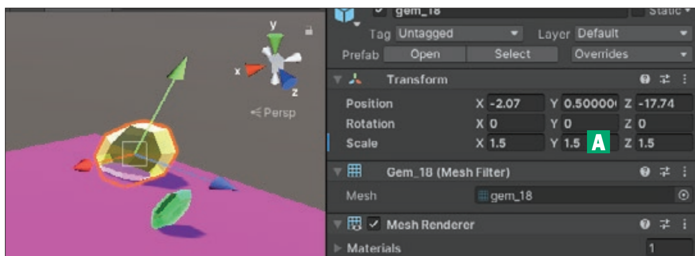
12 Jeśli uruchomimy rozgrywkę, zobaczymy, że czas **K** odlicza się w jej oknie.



Meta

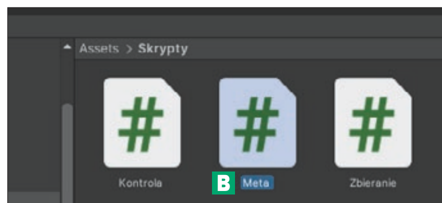
By naszą grę dało się wygrać, na scenie powinna pojawić się meta. Gracz, dochodząc do niej, powinien mieć zebrane już wcześniej wszystkie diamenty. Gdy dojdzie do mety, odliczony czas powinien się zatrzymać – a czas uzyskany przez gracza będzie reprezentował nasz wynik – im niższy, tym lepiej.





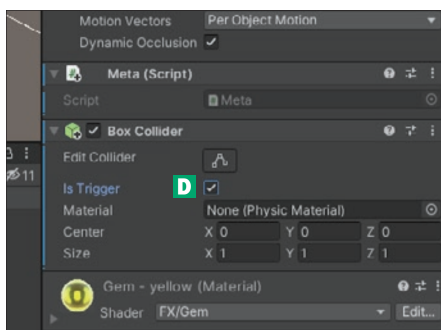
1 By jakaś meta pojawiła się na scenie, możemy ponownie skorzystać z paczki zasobów z diamentami dodanej wcześniej do projektu. Tym razem jako metę można dodać też diament – ale z innej kategorii kolorystycznej, by wyróżniał się od pozostałych. Gdy dodamy taki diament na scenę, możemy zmienić jego skalę na każdej z osi na **1.5** **A** – w ten sposób powiększymy diament pełniący rolę mety.

2 Taki diament powinien też dostać swój skrypt, przez który sprawdzimy, czy gracz go znalazł, i prześlemy do kontrolera informację o końcu gry. Tworzymy zatem nowy skrypt o nazwie **Meta** **B**.



3 Przeciągamy skrypt na obiekt, by wyświetlił się on we właściwościach diamentu-mety **C**.

4 Jeśli mowa o właściwościach tego obiektu, to w nich należy dodać do obiektu komponent **Box Collider** i zaznaczyć w nim opcję **Is Trigger** **D**, by możliwe było wykrywanie wejścia gracza w metę.



5 Nowy skrypt w dużej części będzie przypominał skrypt wykonywany przez pozostałe diamenty. Usuńmy w nim funkcje **Start** i **Update**, by zastąpić je funkcją **OnTriggerEnter**, którą możemy w całości skopiować w miejsce ze skryptu o nazwie **Zbieranie**.

6 We wklejonej funkcji należy usunąć linię **GameObject.FindWithTag("GameController").GetComponent<Kontrola>().zebrane_diamenty += 1;** **E**. Utworzony w ten sposób skrypt będzie sprawiał, że po wejściu gracza w metę – meta zniknie.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Meta : MonoBehaviour
{
    void OnTriggerEnter(Collider other)
    {
        if (other.tag == "Player")
        {
            Destroy(gameObject);
          E
        }
    }
}
```

7 W miejscu usuniętej linii kodu dodajemy teraz deklarację zmiennej o nazwie **k**, która będzie służyła w tym skrypcie do odnoszenia się do kontrolera gry – co uzyskamy, pisząc: **GameObject k = GameObject.FindWithTag("GameController");** **F**.

```
if (other.tag == "Player")
{
    Destroy(gameObject);
F   GameObject k = GameObject.FindWithTag("GameController");
}
```

8 Mając odniesienie do obiektu, możemy dalej utworzyć instrukcję warunkową, poprzez którą będziemy sprawdzać, czy w momencie wejścia gracza na metę posiadał on odpowiednią liczbę zebranych diamentów. Jeśli mamy na scenie 10 diamentów, możemy ustalić, że zebranie ich wszystkich powoduje wygraną lub na przykład już zebranie 8 będzie ją powodowało. W naszej przykładowej grze jest 5 diamentów, stąd w warunku zapis: **k.GetComponent<Kontrola>().zebrane_diamenty >= 5** **G**.

```
    GameObject k = GameObject.FindWithTag("GameController");
    if (k.GetComponent<Kontrola>().zebrane_diamenty >= 5) G
    {

    }
}
```

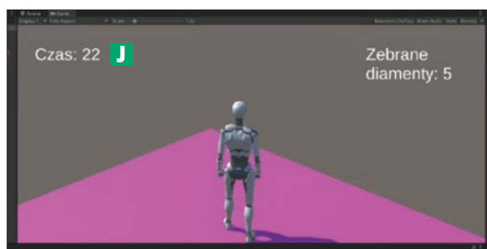
9 W instrukcji warunkowej powinniśmy teraz odnieść się ponownie do kontrolera gry i zmienić stan rozgrywki na wygraną – co zrobimy, pisząc: **k.GetComponent<Kontrola>().stan_gry = "wygrana";** **H**.

```
if (k.GetComponent<Kontrola>().zebrane_diamenty >= 5)
{
H   k.GetComponent<Kontrola>().stan_gry = "wygrana";
}
```

10 Skrypt w tej postaci działa jednak tak, że diament-meta znika zawsze gdy obiekt gracza w niego wejdzie, a stan rozgrywki na wygraną zmieni się tylko wtedy, gdy zbierzemy wszystkie diamenty. To oznacza, że po wejściu w metę, gdy nie mamy jeszcze odpowiedniej liczby diamentów, nie można wrócić do ich zbierania i ponownie wejść w metę, ponieważ meta przestaje istnieć już przy pierwszym wejściu w nią. By temu zaradzić, powinniśmy tak zmienić skrypt, by meta zniknęła tylko wtedy, gdy mamy odpowiednią liczbę diamentów. Zatem instrukcję **Destroy(gameObject)** **I** przenosimy do wnętrza najnowszej instrukcji warunkowej.

```
if (other.tag == "Player")
{
    GameObject k = GameObject.FindWithTag("GameController");
    if (k.GetComponent<Kontrola>().zebrane_diamenty >= 5)
    {
        k.GetComponent<Kontrola>().stan_gry = "wygrana";
    I   Destroy(gameObject);
    }
}
```

11 Gdy teraz dojdziemy do mety, testując rozgrywkę, zobaczymy, że czas **J** zatrzyma się, jednak nie ma jeszcze żadnego



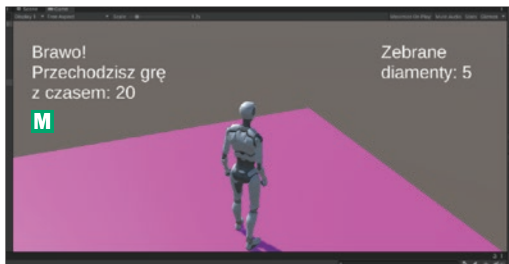
komunikatu, który mówiłby graczowi, że rozgrywka jest przez niego wygrana.

12 Taki komunikat może pojawić się na obiekcie tekstowym do wyświetlania czasu. Jednak by go dodać, powinniśmy jego wyświetlanie umieścić w skrypcie kontrolera gry. Powinna się tam znaleźć kolejna instrukcja warunkowa, która sprawdzi wartość pola **stan_gry** – czy jest ona równa **wygrana** K.

```
            czas_gry += 1;
        }
        txtCzas.text = "Czas: " + czas_gry.ToString();
    }
    if (stan_gry == "wygrana") K
    {

    }
}
```

13 Gdy warunek jest spełniony – możemy dodać instrukcję zmieniającą tekst na obiekcie do wyświetlania czasu – tak by oprócz zatrzymanego czasu pojawiła się tam informacja o ukończeniu gry. Cała linia może mieć postać: **txtCzas.text = "Brawo! Przechodzisz grę z czasem: " + czas_gry.ToString();** L.



14 Możemy przetestować grę. Jeśli dojdziemy do mety, zobaczymy komunikat z informacją o przejściu całej gry M.

```
if (stan_gry == "wygrana")
{
    txtCzas.text = "Brawo! Przechodzisz grę z czasem: " + czas_gry.ToString(); L
}
```

Przegrana

Do dopełnienia naszej gry brakuje jeszcze możliwości jej przegrania. Gracz przegra, jeśli spadnie z platformy. Powinniśmy zatem dorobić w naszej grze mechanizm sprawdzający, czy gracz spadł z platformy.

1 Nasz mechanizm będzie opierał się na sprawdzaniu pozycji Y obiektu gracza. By się do niej odnieść, możemy wyłapać obiekt gracza poprzez przypisany do niego automatycznie tag – **Player**. Samą pozycję na osi Y otrzymamy, zapisując więc **GameObject.FindWithTag("Player").transform.position.y** A. Im mniejszą pozycję Y ma obiekt, tym jest niżej na scenie. Pozycją, na której umieszczone są płaszczyzny pełniące rolę platform, powinno być 0. Zatem jeśli gracz będzie niżej od nich – możemy mówić, że spadł z platform. Jednak gdy gracz chodzi po platformie, jest w animowanym ruchu, który może sprawić, że w pewnym momencie jego pozycja będzie odczytana jako mniejsza od zera, mimo że wciąż jest na platformie. Ta ujemna wartość będzie bardzo

```
        txtCzas.text = "Czas: " + czas_gry.ToString();
    A   if(GameObject.FindWithTag("Player").transform.position.y <= -1)
        {

        }
}
if (stan_gry == "wygrana")
```

mała, jednak może wystąpić. Bezpieczniej będzie zatem, jeśli zbudujemy instrukcję warunkową sprawdzającą, czy pozycja gracza spadła na osi Y poniżej wartości -1 B. Co istotne – takie sprawdzanie powinno odbywać się w funkcji **Update** kontrolera gry, wtedy kiedy spełniony jest warunek mówiący o tym, że trwa rozgrywka.

```
    txtCzas.text = "Czas: " + czas_gry.ToString();
B   if(GameObject.FindWithTag("Player").transform.position.y <= -1)
    {

    }
}
if (stan_gry == "wygrana")
```

```
if(GameObject.FindWithTag("Player").transform.position.y <= -1)
{
    stan_gry = "przegrana"; C
}
```

```
    {
        txtCzas.text = "Brawo! Przechodzisz grę z czasem: " + czas_gry.ToString();
    }
    if (stan_gry == "przegrana") D
    {

    }
}
```

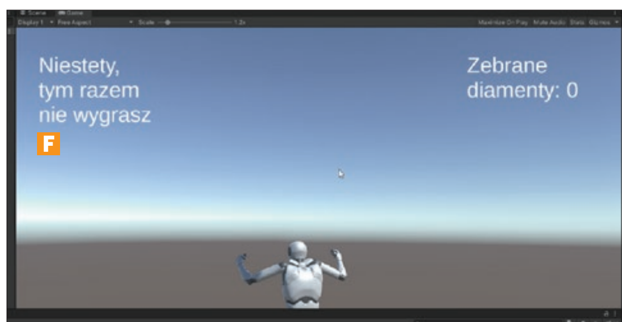
```
if (stan_gry == "przegrana")
{
E txtCzas.text = "Niestety, tym razem nie wygrasz";
}
```

2 Gdy gracz spadnie z platformy, możemy zmienić wartość w polu **stan_gry** na **przegrana** C.

3 Musimy jednak dodać w skrypcie kontrolera gry obsługę takiego stanu gry. W tej samej funkcji należy dodać kolejną instrukcję warunkową, w której warunkiem będzie tym razem **stan_gry == "przegrana"** D.

4 Jeśli wspomniany warunek jest spełniony, należałoby zmienić napis na obiekcie tekstowym wyświetlającym czas, tak by pojawiła się tam informacja o tym, że tym razem nie uda się graczowi wygrać – co można uzyskać, umieszczając w skrypcie: **txtCzas.text = "Niestety, tym razem nie wygrasz";** E.

5 Po tak wprowadzonych zmianach naszą grę można nie tylko wygrać, ale też przegrać. Przetestujmy ją ponownie – jednak tym razem celowo zeskoczmy z platformy, by sprawdzić, czy obiekt tekstowy wyświetli przy spadaniu odpowiedni komunikat F.



CO DALEJ? MOŻEMY ROZBUDOWAĆ PROJEKT!

To już koniec pracy nad naszym ostatnim projektem. Może to być jednak początek dużej przygody z Unity.

By zacząć samemu uczyć się Unity, możemy rozbudować jeszcze ten ostatni projekt.

Postarajmy się na przykład zmienić efekt, jaki widzi gracz w wyniku przegranej i wygranej rozgrywki. Obecnie są to jedynie informacje tekstowe, jednak można przebudować projekt w taki sposób, by stworzyć oddzielne sceny, które byłyby wyświetlane graczowi w zależności od tego, czy ten przegra, czy wygra rozgrywkę.

W przypadku wygranej mogłaby to być scena zawierająca kolejny poziom z inaczej ułożonymi platformami, których mogłoby być więcej, a przejście pomiędzy nimi mogłoby być trudniejsze.

W przypadku przegranej mogłaby to być scena zawierająca **Canvas** i grafikę informacyjną.

A jak zaprogramować przejście do zupełnie innej sceny w projekcie?

Warto przypomnieć sobie początek rozdziału poświęconego grze ze statkiem kosmicznym. Stworzyliśmy tam funkcję, która wykonywała się w momencie kliknięcia w menu na przycisk startujący grę. Jej zawartość to polecenie uruchamiające inną scenę – będzie dla nas przydatne także w tym wypadku.

DŹWIĘKI WE WSZYSTKICH PROJEKTACH

W **rozdziale 3** przeczytaliśmy, jak wprowadzić do projektu obsługę dźwięków. Możemy wykorzystać te informacje do kontynuowania pracy nad wszystkimi trzema grami zrealizowanymi w ramach kolejnych rozdziałów tej książki.

■ W projekcie z **rozdziału 5** można dodać dźwięk wybuchu. Powinien być on odtworzony, gdy asteroida uderza w statek kosmiczny, czyli w momencie, kiedy pojawia się również wybuch. To oznacza,

```
if (other.tag == "Player") A
{
    Debug.Log("Trafiam w statek"
```

że należy użyć instrukcji odtwarzającej dźwięk wewnątrz instrukcji warunkowej **A** z funkcji **OnTriggerEnter**.

■ W projekcie z **rozdziału 6** można dodać dźwięk tykającego zegara bomby, odtwarzany z każdą upływającą sekundą.

```
if (odliczony >= 1f)
{
    czas = czas - 1; B
```

Zatem instrukcja odtwarzająca dźwięk powinna być dodana w miejscu, w którym zmienia się wartość czasu **B**.

■ Projekt z **rozdziału 7** także może być rozbudowany o obsługę dźwięków. Może się tu pojawić wybrany przez nas efekt dźwiękowy – może on być odtwarzany, gdy gracz zdobędzie diament dla podkreślenia wygranej.



Fot. Bluebay/123rf.com

JAK SKORZYSTAĆ Z E-WYDANIA KSIĄŻKI

W KŚ+ znajdziemy **e-wydanie tej Biblioteczki, obraz ISO dołączonej do niej płyty z narzędziami do tworzenia gier oraz plik PDF książki do pobrania.**

1 Otwieramy stronę **ksplus.pl**. Logujemy się **A** (używamy konta z serwisu **Komputerswiat.pl**). Jeżeli nie mamy konta, klikamy na **B**, by się zarejestrować.

B Załóż konto Logowanie A
Zarejestruj kod

2 Po zalogowaniu się możemy zarejestrować kod nadrukowany na płycie dołączonej do książki. Wystarczy kliknąć na **C** i przepisać kod.

Moje konto
C Zarejestruj kod

3 Uzyskamy w ten sposób dostęp do e-wydania **D** i do bonusowego obrazu płyty **E**. Do serwisu KŚ+ możemy logować się z dowolnego urządzenia z dostępem do internetu.

CZYTAJ E-WYDANIE D
PROGRAMY E
BONUSY

UWAGA! W KŚ+ ZA DARMO E-WYDANIE KSIĄŻKI ORAZ PLIK ISO PŁYTY

POLECAMY INNE NASZE KSIĄŻKI



100 TRIKÓW RATUNKOWYCH

Rozwiązania najczęstszych problemów z systemem, Wi-Fi, programami i sprzętem – zarówno proste, jak i zaawansowane wskazówki krok po kroku. Na DVD: najlepsze darmowe narzędzia serwisowe.



WSZYSTKO O PYTHONIE

Pigułka wiedzy o Pythonie: instrukcje warunkowe, pętle, definiowanie funkcji, korzystanie z klas, importowanie modułów, komunikacja z internetem. Na DVD: pliki szkoleniowe, narzędzia ułatwiające programowanie.



Konrad Jagaciak
programista prowadzący zajęcia komputerowe z młodzieżą

ZOSTAŃ TWÓRCĄ GIER!

Unity to jeden z najpopularniejszych silników, czyli kompleksowych rozwiązań, w których powstają doskonałe gry komputerowe i na smartfony. Dzięki wbudowanym mechanizmom i bogatym paczkom zasobów jesteśmy w stanie tworzyć w Unity – stosunkowo niewielkim nakładem pracy – rozbudowane gry z trójwymiarową grafiką.

Opisane w tej książce projekty w dużej mierze bazują na trójwymiarowej grafice z darmowych zasobów z Asset Store. Umożliwiają one tworzenie gier atrakcyjnych graficznie bez konieczności uczenia się obsługi narzędzi do tworzenia trójwymiarowych modeli graficznych. To sprawia, że większą wagę możemy przyłożyć do samego budowania świata gry z gotowych elementów, a także do programowania.

Książka zawiera zarówno ogólną wiedzę na temat obsługi edytora Unity i programowania w C#, jak i opisy krok po kroku, jak tworzyć przykładowe gry.

Wszystkie opisane projekty znajdziemy na dołączonej do książki płycie. DVD zawiera też Unity Hub oraz zestaw najbardziej znanych darmowych silników gier i narzędzi programistycznych.

CENA 16,90 ZŁ
W TYM 5% VAT

Płyta DVD jest dodatkiem do książki

ISBN 978-83-8250-101-8 INDEKS 321 958



Nr 6/2021 (116)



KOMPUTER ŚWIAT BIBLIOTECZKA